



# PROCEEDINGS

and Additional Papers

## C++ Workshop

Santa Fe, NM, 1987

Proceedings USENIX C++ Workshop 1987

For additional copies of these proceedings write

USENIX Association  
P.O. Box 2299  
Berkeley, CA 94710 USA

The price is \$20.

Overseas postage per copy is \$25 for air or \$5 for surface.

SECOND PRINTING, 1988

Copyright © 1987 USENIX Association  
All Rights Reserved

This volume is published as a collective work.  
Rights to individual papers remain  
with the author or the author's employer.

UNIX is a registered trademark of AT&T.  
Other trademarks are noted in the text.

**USENIX Association**

**C++ Workshop**

**Santa Fe, NM**

**PROCEEDINGS**

**November 9-10, 1987**

# ACKNOWLEDGMENTS

Sponsored by: USENIX Association  
P.O. Box 2299  
Berkeley, CA 94710

Program Chair: Keith Gorlen National Institutes of Health

Workshop held at: Eldorado Clarion Hotel  
Sante Fe, New Mexico

USENIX Meeting Planner: Judith F. DesHarnais

USENIX Board Liasion: David A. Yost

Proceedings Production: Peter H. Salus  
Tom Strong USENIX Executive Director  
Strong Consulting

# TABLE OF CONTENTS

Introduction .....	v
The Evolution of C++ 1985 to 1987 .....	1
<i>Bjarne Stroustrup</i>	
Two Extensions to C++: A Dynamic Link Editor and Inner Data .....	23
<i>Philippe Gautron and Marc Shapiro</i>	
The Architecture of a C++ Compiler .....	35
<i>Steve Dewhurst</i>	
C++ for OS/2 .....	47
<i>John Carolan</i>	
C++ on the Macintosh .....	67
<i>Ken Friedenbach</i>	
Extending the C++ Task System for Real-Time Control .....	77
<i>Jonathan E. Shopiro</i>	
C++ on a Parallel Machine .....	95
<i>Thomas W. Doepfner, Jr. and Alan J. Gebele</i>	
The Design of a Multiprocessor Operating System .....	109
<i>Roy Campbell, Vincent Russo and Gary Johnston</i>	
C*: A C++-like Language for Data-Parallel Computation .....	127
<i>John R. Rose</i>	
Implementing a Compiler in C++ .....	135
<i>John R. Rose</i>	
A Style for Writing C++ Classes .....	147
<i>Peter Kirsliis</i>	
Extending Stream I/O to include Formats .....	149
<i>Mark Rafter</i>	
What is "Object-Oriented Programming"? .....	159
<i>Bjarne Stroustrup</i>	
An Object-Oriented Class Library for C++ Programs .....	181
<i>Keith E. Gorlen</i>	
Object-Oriented Class Library for C++ .....	209
<i>Ken Fuhrman</i>	

Teaching C++ .....	232
<i>Tsvi Bar-David</i>	
Modelling Graphical Data with C++ .....	238
<i>Al Conrad</i>	
Integrated Class Structures for Image Pattern Recognition and Computer Graphics ....	240
<i>James M. Coggins</i>	
Using C++ to Develop a WYSIWYG Hypertext Toolkit .....	246
<i>Jim Waldo</i>	
The Design and Implementation of InterViews .....	256
<i>Mark A. Linton and Paul R. Calder</i>	
The Design of the Allegro Programming Environment .....	268
<i>Mark A. Linton, Russell W. Quong and Paul R. Calder</i>	
A C++ Class Browser .....	274
<i>Raghunath Raghavan, Niranjana Ramakrishnan and Sue Strater</i>	
Pi: A Case Study in Object-Oriented Programming .....	282
<i>Tom Cargill</i>	
CLAM — an Open System for Graphical User Interfaces .....	305
<i>Lisa A. Call, David L. Cohrs and Barton P. Miller</i>	
Experience in Using C++ for Software System Development .....	327
<i>William E. Hopkins</i>	
Program Translation by Manipulating Abstract Syntax Trees .....	345
<i>Xing Liu and Patrick Gonley</i>	
C*: An Extended C Language .....	361
<i>John R. Rose and Guy L. Steele, Jr.</i>	
Possible Directions for C++ .....	399
<i>Bjarne Stroustrup</i>	
A Set of C++ Classes .....	417
<i>Bjarne Stroustrup and Jonathan E. Shapiro</i>	
C++ vs. Lisp .....	440
<i>Howard Trickey</i>	
Avalon/C++ .....	451
<i>David Detlefs, Maurice Herlihy, Karen Kietzke, and Jeannette Wing</i>	
Attendee List .....	461

## INTRODUCTION

The volume in your hands is substantially heftier than any other USENIX workshop proceedings. There are two reasons for this: the first is that Keith Gorlen, the C++ Workshop's program chair, was extraordinarily active in prodding the various presenters to get their papers in (the direct result of this is that only one paper actually presented in Santa Fe is not included); the second reason is that there are several important C++ papers grouped together at the end of this volume which were not presented in Santa Fe.

There is no accessible source for these important C++ papers and Dave Yost, who was the instigator of the workshop and served as the USENIX Board's liaison for it, thought that their inclusion would make the volume a yet more important contribution.

The C++ Workshop was the largest and best-attended workshop that USENIX has sponsored. A direct result of this is that there are plans for a mini-conference concerning C++ for October 1988, with Denver, CO, the most likely location. If you are not on the USENIX mailing list and are interested in this C++ Conference, which will include both tutorials and technical sessions, please contact the USENIX Conference Office (`{uunet, ucbvax}/usenix!judy` or P.O. Box 385; Sunset Beach, CA 90742; 213-592-1381).

One way to ensure your continuing technical awareness is through membership in the USENIX Association. The Association holds two large technical conferences and several workshops each year (four in 1987, of which C++ was one); publishes the technical journal, *Computing Systems*, quarterly; and publishes a bimonthly newsletter, *login*:. The USENIX Association is also the distributor of 2.10BSD tapes and 4.3BSD manuals. For membership information, phone the Association office at 415-528-8649 or write `{uunet, ucbvax}/usenix!office`.

Peter H. Salus  
Executive Director

## NOTES

Lisa A. Call, et al, "CLAM," appeared in Proceedings of the OOPSLA '87 Conference, Orlando, FL, October 1987, pp 277-286. It also appears as University of Wisconsin Computer Sciences Department Technical Report #691, April '87.

William E. Hopkins, "Experience...," describes work performed between mid-1985 and early 1986; it was last modified in March 1987.

Rose and Steel, "C\*," originally appeared in the May 1987 Proceedings of the International Supercomputer Institute and is reprinted with premission of the Institute and the authors. It also appeared as a Thinking Machines Technical Report.

Bjarne Stroustrup, "What is 'Object-Oriented Programming'?" originally appeared in Proceedings ECOOP'87, Paris, June 1987. It also appears in Springer Verlag, Lecture Notes in Computer Science, Vol 276, pp 51-70. It was most recently revised, for this publication, on December 11, 1987.

Stroustrup, "Possible Directions for C++," was previously presented at the AT&T C++ users' group meeting, September 1987.

# The Evolution of C++: 1985 to 1987

Bjarne Stroustrup

## ABSTRACT

*The C++ Programming Language*<sup>5</sup> describes C++ as defined and implemented in August 1985. This paper describes the growth of the language since then and clarifies a few points in the definition. It is emphasized that these language modifications are extensions; C++ has been and will remain a stable language suitable for long term software development. The main new features of C++ are: multiple inheritance, recursive definition of assignment and initialization, class specific new and delete operators, protected members, overloading of operator -, >, and pointers to members.

## Introduction

As promised in *The C++ Programming Language*<sup>5</sup>, C++ has been evolving to meet the needs of its users. This evolution has been guided by the experience of users of widely varying backgrounds working in a great range of application areas. The primary aim of the extensions has been to enhance C++ as a language for data abstraction and object-oriented programming<sup>7</sup> in general and to enhance it as a tool for writing high quality libraries of user-defined types in particular. By a high-quality library I mean a library that provides a concept to a user in the form of one or more classes that are convenient, safe, and efficient to use. In this context, *safe* means that a class provides a specific type-secure interface between the users of the library and its providers; *efficient* means that use of the class does not impose large overheads in run-time or space on the user compared with "hand written C code."

Portability of at least some C++ implementations is a key design goal†. Consequently, extensions that would add significantly to the porting time or significantly to the demands on resources for a C++ compiler have been avoided. This ideal of language evolution can be contrasted with plausible alternative directions such as making programming convenient

- at the expense of efficiency or structure;

- for novices at the expense of generality;

- in a specific application area by adding special purpose features to the language;

- by adding language features to increase integration into a specific C++ environment.

For some ideas of where these ideas of language evolution might lead C++ see references 7 and 8.

A programming language is only one part of a programmer's world. Naturally, work is being done in many other fields (such as tools, environments, libraries, education and design methods) to make C++ programming more pleasant and effective. This paper, however, deals strictly with language and language implementation issues.

The features described are in use and the C++ implementation that supports them will become generally available within a few months.

† One day is a reasonable time for porting the current implementation of C++ to a new system.

## 1 protected Members

The simple private/public model of data hiding served C++ well where C++ was used essentially as a data abstraction language and for a large class of problems where inheritance was used for object-oriented programming. However, when derived classes are used there are two kinds of users of a class: "the general public" and "derived classes." The members and friends that implement the operations on the class operate on the class objects on behalf of these users. The private/public mechanism allows the programmer to distinguish clearly between the implementors and the general public, but does not provide a way of catering specifically for derived classes<sup>†</sup>. This often caused the data hiding mechanisms to be ignored:

```
class X {    // One bad way:
...
public:
    int a;    // "a" should have been private
              // don't use it unless you are a member of a derived class
...
};
```

Another symptom of this problem was overuse of friend declarations:

```
class X {    // Another bad way:
friend class D1;    // make derived classes friends
friend class D2;    // to give access to private member "a"
...
friend class Dn;
...
    int a;
public:
    ...
};
```

The solution adopted was to introduce the concept of protected members. A protected member is accessible to members and friends of a derived class as if it were public, but inaccessible to "the general public" just like private members. For example:

```
class X {
// private by default:
    int priv;
protected:
    int prot;
public:
    int publ;
};

class Y : public X {
    void mf();
};

Y::mf()
{
    priv = 1;    // error: priv is private
    prot = 2;    // OK: prot is protected and mf2() is a member of Y
    publ = 3;    // OK: publ is public
}
```

<sup>†</sup> An interesting discussion of access and encapsulation problems in languages with inheritance mechanisms can be found in reference 4.

```

void f(Y* p)
{
    p->priv = 1; // error: priv is private
    p->prot = 2; // error: prot is protected and f() is not a friend
                // or a member of X or Y
    p->publ = 3; // OK: publ is public
}

```

A more realistic example of the use of protected can be found in section 3.

A friend function has the same access to protected members as a member function.

A subtle point is that accessibility of protected members depends on the static type of the pointer used in the access. A member or a friend of a derived class has access only to protected members of objects that are known to be of its derived type. For example:

```

class Z : public Y {
    ...
};

void Y::mf()
{
    prot = 2; // OK: prot is protected and mf() is a member

    X a;
    a.prot = 3; // error: prot is protected and a is not a Y

    X* p = this;
    p->prot = 3; // error: prot is protected
                // and p is not a pointer to Y

    Z b;
    b.prot = 4; // OK: prot is protected
                // and mf() is a member and a Z is a Y
}

```

## 2 Access Control

The syntax for expressing access control concerns has been made more flexible.

The following example confuses most beginners and even experts occasionally get bitten:

```

class X {
    ...
public:
    int f();
};

class Y : X { ... };

int g(Y* p)
{
    ...
    return p->f();
};

```

Here X is by default declared to be a private base class of Y. This means that X is not a sub-type of Y so that the call `p->f()` is illegal because Y does not have a public function `f()`. Private base classes are quite an important concept, but to avoid confusion it is recommended that they be declared explicitly private:

```

class Y : private X { ... };

```

Several public, private, and protected sections are allowed in a class declaration:

```

class X {
public:
    int i1;
private:
    int i2;
public:
    int i3;
};

```

These sections can appear in any order. This implies that it is possible to adopt a style where the public interface of a class appears textually before the private "implementation details":

```

struct S {
// public by default:
    f();
    int i1;
    ...
private:
    g();
    int i2;
    ...
};

```

It was always possible to specify that a member of a private base should be considered public:

```

class X {
    ...
public:
    f1();
    f2();
};

class Y : private X {
    ...
public:
    X::f1;    // f1 is a public member of Y
              // (but f2 is still private)
};

```

The complementary operation, making a member of a public base private, is also allowed:

```

class X {
    ...
public:
    f1();
    f2();
};

class Y : public X {
    ...
private:
    X::f2;    // f2 is a private member of Y
              // (but f1 is still public)
};

```

Note that making `f2()` private implies that `X` is no longer a sub-type of `Y`; that is, there is no implicit coercion of a pointer to `Y` to a pointer to `X` and public members of `X` are not automatically public members of `Y`.

### 3 Multiple Inheritance

Consider writing a simulation of a network of computers. Each node in the network is represented by an object of class `Switch`, each user or computer by an object of class `Terminal`, and each communication line by an object of class `Line`. One way to monitor the simulation (or a real network of the same structure) would be to display the state of objects of various classes on a screen. Each object to be displayed is represented as an object of class `Displayed`. Objects of class `Displayed` are under control of a display manager that ensures regular update of a screen and/or data base. The classes `Terminal` and `Switch` are derived from a class `Task` that provides the basic facilities for co-routine style behavior. Objects of class `Task` are under control of a task manager (scheduler) that manages the real processor(s).

Ideally `Task` and `Displayed` are classes from a standard library. If you want to display a terminal, class `Terminal` must be derived from class `Displayed`. Class `Terminal`, however, is already derived from class `Task`. In a single inheritance language, such as `Simula67`, we have only two ways of solving this problem: deriving `Task` from `Displayed` or deriving `Displayed` from `Task`. Neither is ideal since they both create a dependency between the library versions of two fundamental and independent concepts. Ideally one would want to be able choose between saying that a `Terminal` is a `Task` *and* a `Displayed`; that a `Line` is a `Displayed` *but not* a `Task`; and that a `Switch` is a `Task` *but not* a `Displayed`.

The ability to express this using a class hierarchy, that is, to derive a class from more than one base class, is usually referred to as *multiple inheritance*. Other examples involve the representation of various kinds of windows in a window system<sup>9</sup> and the representation of various kinds of processors and compilers for a multi-machine, multi-environment debugger<sup>1</sup>.

In general, multiple inheritance allows a user to combine concepts represented as classes into a composite concept represented as a derived class. C++ allows this to be done in a general, type-safe, compact, and efficient manner. The basic scheme allows independent concepts to be combined and ambiguities to be detected at compile time. An extension of the base class concept, called *virtual base classes*, allows dependencies between classes in an inheritance DAG (Directed Acyclic Graph) to be expressed.

Ambiguous uses are detected at compile time:

```
struct A { f(); ... };
struct B { f(); ... };
struct C : A, B { ... };

void g() {
    C* p;
    p->f(); // error: ambiguous
}
```

Note that it is not an error to combine classes containing the same member names in an inheritance DAG. The error only occurs when a name is used in an ambiguous way and only then does the compiler have to reject the program. This is important since most potential ambiguities in a program never appear as actual ambiguities. Considering a potential ambiguity an error would be far too restrictive†.

Typically one would resolve the ambiguity by adding a function:

† The strategy for dealing with ambiguities in inheritance DAGs is essentially the same as the strategy for dealing with ambiguities in expression evaluation involving overloaded operators and user-defined coercions.

Note that the public/private/protected access controls do not affect the ambiguity control mechanisms. Had only `A::f()` been public the call `p->f()` would still be considered ambiguous.

```

struct C : A, B {
    f()
    {
        // C's own stuff
        A::f();
        B::f();
    }
    ...
}

```

This example shows the usefulness of the C++ way of naming members of a base class explicitly with the name of the base class. In the restricted case of single inheritance, this way is marginally less elegant than the approach taken by Smalltalk and other languages (simply referring to "my super class" instead of using an explicit name). However, the C++ approach extends cleanly to multiple inheritance.

A class can appear more than once in an inheritance DAG:

```

class A : public L { ... };
class B : public L { ... };
class C : public A, public B { ... };

```

In this case, an object of class C has two sub-objects of class L: A::L and B::L. This is often useful, as in the case of an implementation of lists requiring each element on a list to contain a link element. If in the example above L is a link class then a C can be on both the list of As and the list of Bs at the same time.

Virtual functions work as expected:

```

struct A { virtual f(); ... };
struct B { virtual g(); ... };
struct C : A, B { f(); g(); ... };

void ff()
{
    C obj;
    A* pa = &obj;
    B* pb = &obj;

    pa->f(); // calls C::f
    pb->g(); // calls C::g
}

```

This way of combining classes is ideal for representing the union of independent or nearly independent concepts. However, in some interesting cases, such as the window example, a more explicit way of expressing sharing and dependency is needed.

Virtual base classes provide a mechanism for sharing between sub-objects in an inheritance DAG and for expressing dependencies among such sub-objects:

```

class A : public virtual W { ... };
class B : public virtual W { ... };
class C : public A, public B, public virtual W { ... };

```

In this case there is only one object of class W in class C.

Constructing the tables for virtual function calls can get quite complicated when virtual base classes are used. However, virtual functions work as expected:

```

class W {
public:
    ...
    virtual void f();
    virtual void g();
    virtual void h();
    virtual void k();
    ...
};

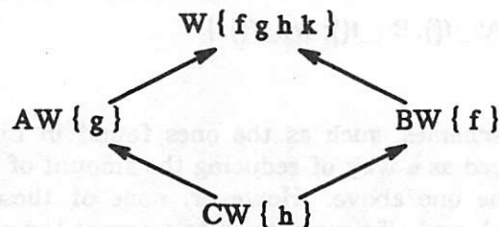
class AW : public virtual W { void g(); ... };
class BW : public virtual W { void f(); ... };
class CW : public AW , public BW , public virtual W { void h(); ... };

CW* pcw = new CW;

pcw->f();           // invokes BW::f()
pcw->g();           // invokes AW::g()
pcw->h();           // invokes CW::h()
((AW*)pcw)->f();    // invokes BW::f() !!!

```

Ambiguities are easily detected at the point where CW's table of virtual functions is constructed. The rule for detecting ambiguities in a class DAG is that all re-definitions of a virtual function from a virtual base class must occur on a single path through the DAG. The example above can be drawn like this:



Note that a call "up" through one path of the DAG to a virtual function may result in the call of a function (re-defined) in another path (as happened in the call `((AW*)pcw)->f()` in the example above). In this example, an ambiguity would occur if a function `f()` was added to AW. This ambiguity might be resolved by adding a function `f()` to CW.

Programming using virtual bases is a bit trickier than programming using non-virtual bases. The problem is to avoid multiple calls of a function in a virtual class when that is not desired. Here is a possible style. Let each class provide a protected function doing "its own stuff", `_f()`, and a public function `f()` doing "its own stuff" by calling `_f()` and calling the `_f()`s for its base classes:

```

class W {
protected:
    ...
    _f() { my stuff }
    ...
public:
    f() { _f(); }
    ...
};

```

```

class A : public virtual W {
    ...
protected:
    __f() { my stuff }
    ...
public:
    f() { __f(); W::__f(); }
    ...
};

class B : public virtual W {
    ...
protected:
    __f() { my stuff }
    ...
public:
    f() { __f(); W::__f(); }
    ...
};

class C : public A, public B, public virtual W {
    ...
protected:
    __f() { my stuff }
    ...
public:
    f() { __f(); A::__f(); B::__f(); W::__f(); }
    ...
};

```

Method combination schemes, such as the ones found in Lisp systems with multiple inheritance, were considered as a way of reducing the amount of code a programmer needed to write in cases like the one above. However, none of these schemes appeared to be sufficiently simple, general, and efficient enough to warrant the complexity it would add to C++.

As described in reference 6 a virtual function call is about as efficient as a normal function call even in the case of multiple inheritance. The added cost is 5 to 6 memory references per call. This compares with the 3 to 4 extra memory references incurred by a virtual function call in a C++ compiler providing only single inheritance. The multiple inheritance scheme currently used causes an increase of about 50% in the size of the tables used to implement the virtual functions compared with the older single inheritance implementation. To offset that, the multiple inheritance implementation optimizes away quite a few spurious tables generated by the older single-inheritance implementations so that the memory requirement of a program using virtual functions actually decreases in most cases.

It would have been nice if there had been absolutely no added cost for the multiple inheritance scheme when only single inheritance is used. Such schemes exist, but involve the use of tricks that cannot be done by a C++ compiler generating C.

#### 4 Base and Member Initialization

The syntax for initializing base classes and members has been extended to cope with multiple inheritance and the order of initialization has been more precisely defined. Leaving the initialization order unspecified in the original definition of C++ gave an unnecessary degree of freedom to language implementors at the expense of the users. In most cases, the order of initialization of members doesn't matter and in most cases where it does matter, the order dependency is an indication of bad design. In a few cases, however, the programmer absolutely needs control of the order of initialization. For example, consider transmitting objects between machines. An object must be re-constructed by a receiver in exactly

the reverse order in which it was decomposed for transmission by a sender. This can not be guaranteed for objects communicated between programs compiled by compilers from different suppliers unless the language specifies the order of construction.

Consider:

```
class A { ... A(int); A(); };
class B { ... B(int); B(); };
```

```
class C : public A, public B {
    const a;
    int& b;
public:
    C(int&);
};
```

In a constructor the sub-objects representing base classes can be referred to by their class names:

```
C::C(int& rr) : A(1), B(2), a(3), b(rr) { ... }
```

The initialization takes place in the order specified by the list of initializers except that base classes are always initialized before members, so in this case the initialization order is A, B, a, b and

```
C::C(int& rr) : b(rr), B(2), a(3), A(1) { ... }
```

gives the initialization order B, A, b, a.

Unmentioned base classes are initialized after the explicitly initialized bases using their default initializers in the order they appear in the class declaration. For example:

```
C::C(int& rr) : b(rr), a(3) { ... }
```

gives the initialization order A, B, b, a and

```
C::C(int& rr) : b(rr), a(3), B(1) { ... }
```

gives the order B, A, b, a.

Using the base class name explicitly even in the single inheritance case makes clearer code:

```
class vector {
    ...
public:
    vector(int);
    ...
};

class vec : public vector {
    ...
public:
    vec(int,int);
    ...
}
```

It is reasonably clear even to novices what is going on here:

```
vec::vec(int h, int l) : vector(h-l-1) { ... }
```

On the other hand, this version:

```
vec::vec(int h, int l) : (h-l-1) { ... }
```

has caused much confusion over the years. The old-style base class initializer is of course still accepted. It can only be used in the single inheritance case since it is ambiguous otherwise.

Virtual base classes constitute a special case. A virtual base class cannot be explicitly initialized. This implies that a class can only be a virtual base provided it has either no constructor at all or a constructor taking no arguments. A virtual base is constructed before any of its derived classes. Virtual bases are constructed before any non-virtual bases and in the order they appear on a depth first left-to-right traversal of the inheritance DAG. This rule applies recursively for virtual bases of virtual bases.

The order of destructor calls is defined to be the reverse order of appearance in the class declaration (members before bases). There is no way for the programmer to control this order except by the declaration order. A virtual base is destroyed after all of its derived classes.

It might be worth mentioning that virtual destructors are allowed:

```
struct B { ... virtual ~B(); };

struct D : B { ~D(); };

void g() {
    B* p = new D;
    delete p; // D::~D() is called
}
```

Virtual destructors have always been allowed and they are very useful but it might be worth while to mention them explicitly anyway.

## 5 Assignment and Initialization

C++ originally had assignment and initialization default defined as bitwise copy of an object. This caused problems in many cases such as when an object of a class with assignment was used as a member of a class that did not have assignment defined:

```
class X {
public:
    ...
    const X& operator=(const X&);
    ...
};

class Y {
    X a;
    ...
};

void f()
{
    Y y1, y2;
    ...
    y1 = y2;
}
```

Assuming that assignment was not defined for Y, y2.a is copied into y1.a using a bitwise copy. This invariably turns out to be an error and the programmer has to add an assignment operator to class Y:

```

class Y {
    X a;
    ...
    const Y& operator=(const Y& arg)
    {
        a = arg.a;
        ...
    }
};

```

To cope with this problem in general, assignment in C++ is now defined as memberwise assignment of non-static members and base class objects.<sup>†</sup> Naturally, this rule applies recursively until a member of a built-in type is found. This implies that for a class X, X(const X&) and const X& X::operator=(const X&) will be supplied where necessary by the compiler, as has always been the case for X::X() and X::~X(). Unless the user supplies it, X(const X&) will be created for a class X that has

- [1] a member or base of class Z for which Z::operator= or Z::Z(Z&) is defined, or
- [2] a virtual function or a virtual base class.

Note that since access controls are correctly applied to both implicit and explicit copy operations we actually have a way of prohibiting assignment of objects of a given class X:

```

class X {
    // Objects of class X cannot be copied
    // except by members of X
    void operator=(X&);
    X(X&);
    ...
public:
    X(int);
    ...
};

void f() {
    X a(1);
    X b = a; // error: X::X(X&) private
    b = a;   // error: X::operator=(X&) private
}

```

The automatic creation of X::X(const X&) and X::operator=(const X&) has interesting implications on the legality of some assignment operations. Note that if X is a public base class of Y then a Y object is a legal argument for a function that requires an X&. In particular:

```

struct X { int aa; };
struct Y : X { int bb; };

void f() {
    X xx;
    Y yy;
    xx = yy; // OK: a Y is an X
              // xx==yy; means xx.operator=((X&)yy);
              // and is optimized to xx.aa = yy.aa
}

```

Defining assignment as memberwise assignment implies that operator=() isn't inherited in the ordinary manner. Instead, the appropriate assignment operator is if necessary

<sup>†</sup> One could argue that the original definition of C++ was inconsistent in requiring bitwise copy of objects of class Y, yet guaranteeing that X::operator=() would be applied for copying objects of a class X. In this case both guarantees cannot be fulfilled.

generated for each class. This implies that the "opposite" assignment of an object of a base class to a variable of a derived class is illegal as ever:

```
void f() {
    X xx;
    Y yy;
    yy = xx; // NO, an X is not a Y
}
```

The extension of the assignment semantics to allow assignment of an object of a derived class to a variable of a public base class had been repeatedly requested by users. The direct connection to the recursive memberwise assignment semantics became clear only through work on the two apparently independent problems.

## 6 Operators new and delete

If a user wanted to take over allocation of objects of a class X the only way used to be to assign to this on each path through every constructor for X. This is a very powerful and general mechanism. It is also non-obvious, error prone, repetitive, too subtle when derived classes are used, and essentially unmanageable when multiple inheritance is used. For example:

```
class X {
    int on_free_store;
    ...
public:
    X();
    X(int i);
    ~X();
    ...
}
```

Every constructor needs code to determine when to use the user-defined allocation strategy:

```
X::X() {
    if (this == 0) {
        this = myalloc(sizeof(X));
        on_free_store = 1;
    }
    else {
        // static or automatic or member of aggregate
        this = this; // forget this assignment at your peril
        on_free_store = 0;
    }
    // initialize
}
```

and the destructor needs code to determine when to use the user-defined de-allocation strategy:

```
X::~X() {
    // cleanup
    if (on_free_store) {
        myfree(this);
        this = 0; // forget this assignment at your peril
    }
}
```

This user-defined allocation and de-allocation strategy isn't inherited by derived classes in the usual way. The fundamental problem with the "assign to this" approach to user-controlled memory management is that initialization and memory management code is intertwined in an ad hoc manner. The alternative is to overload the allocation operator new for class X:

```

class X {
public:
    void* operator new(long sz) { return myalloc(sz); }
    void operator delete(X* p) { myfree(p); }

    X() { /* initialize */ }
    X(int i) { /* initialize */ }

    ~X() { /* cleanup */ }
};

```

Now `X::operator new()` will be used instead of the global operator `new()` for objects of class `X`. Note that this does not affect other uses of operator `new` within the scope of `X`:

```

void* X::operator new(long s)
{
    void* p = new char[s]; // global operator new as usual
    ...
    return p;
}

void X::operator delete(X* p)
{
    ...
    delete (void*) p; // global operator delete as usual
}

```

The usual rules for inheritance apply:

```

class Y : X { ... }; // objects of class Y are also allocated
                    // using X::operator new

```

This is the reason `X::operator new()` needs an argument specifying the amount of store to be allocated; `sizeof(Y)` may be different from `sizeof(X)`. Naturally, a class that is never a base class need not use the size argument:

```

void* Z::operator new(long) { return next_free_Z(); }

```

This optimization should not be used unless the programmer is perfectly sure that `Z` is not used as a base class because if it is disaster will happen.

An operator `new()`, be it local or global, is only used for free store allocation so

```

X a1;

void f()
{
    X a;
    X v[10];
}

```

does not involve any operator `new()`, and

```

X v[10];

void f()
{
    X v[10];
    X* p = new X[10];
}

```

does not involve `X::operator new()`

Like the global operator `new()`, `X::operator new()` returns a `void*`. This indicates that it returns uninitialized memory. It is the job of the compiler to ensure that the memory

returned by this function is converted to the proper type and if necessary initialized using the appropriate constructor. This is exactly what happens for the global operator new(). Note that the this pointer in X::operator new() is un-initialized.

It is occasionally useful to know if an object is allocated using operator new. This too can be done using a local operator new():

```
class X {
    static X* last_X;
    ...
    X();
    void* operator new(long s)
    {
        return last_X = ::operator new(s);
    }
    ...
};

X::X()
{
    if (this == last_X) {
        // on free store
    }
    else {
        // static or automatic or member of aggregate
    }
    ...
}
```

With the introduction of operator new() and operator delete() member functions there do not appear to be any good reasons for assigning to this pointers.

## 7 Operator ->

Until now -> has been one of the few operators a programmer couldn't define. This made it hard to create classes of objects intended to behave like "smart pointers." When overloading, -> is considered a unary operator (of its left hand operand) and -> is reapplied to the result of executing operator->(). Hence the return type of an operator->() function must be a pointer to a class or an object of a class for which operator->() is defined. For example:

```
struct Y { int m; };

class X {
    Y* p;
    ...
    Y* operator->() {
        if (p == 0) {
            // initialize p
        }
        else {
            // check p
        }
        return p;
    }
    ...
};
```

Here, class X is defined so that objects of type X act as pointers to objects of class Y, except that some suitable computation is performed on each access.

```

void f(X x, X& xr, X* xp)
{
    x->m;    // x.p->m
    xr->m;    // xr.p->m
    xp->m;    // error: X does not have a member m
}

```

Like `operator[]()` and `operator()()`, `operator->()` must be a member function (unlike `operator+()`, `operator-()`, `operator<()`, etc. that are often most useful as friend functions).

The dot operator still cannot be overloaded.

For ordinary pointers, use of `->` is synonymous with some uses of unary `*` and `[]`. For example, for

```
Y* p;
```

it holds that:

```
p->m == (*p).m == p[0].m
```

As usual, no such guarantee is provided for user-defined operators. The equivalence can be provided where desired:

```

class X {
public:
    Y* p;
    Y* operator->() { return p; }
    Y& operator*() { return *p; }
    Y& operator[](int i) { return p[i]; }
};

```

If you provide more than one of these operators it might be wise to provide the equivalence exactly as it is wise to ensure that `x+=1` has the same effect as `x=x+1` for a simple variable `x` of some class if `+=`, `=`, and `+` are provided.

The overloading of `->` is important to a class of interesting programs, just like overloading `[]`, and not just a minor curiosity. The reason is that *indirection* is a key concept and that overloading `->` provides a clean, direct, and efficient way of representing it in a program. Another way of looking at operator `->` is to consider it a way of providing C++ with a limited, but very useful, form of *delegation*<sup>2</sup>.

## 8 Pointers to Members

As mentioned in reference 5, it was an obvious deficiency that there was no way of expressing the concept of a pointer to a member of a class in C++. This led to the need to "cheat" the type system in cases, such as error handling, where pointers to functions are traditionally used. Consider this example:

```

struct S {
    int mf(char*);
};

```

The structure `S` is declared to be a (trivial) type for which the member function `mf` is declared. Given a variable of type `S` the function `mf` can be called:

```

S a;
int i = a.mf("hello");

```

The question is "What is the type of `mf`?"

The equivalent type of a non-member function

```
int f(char*);
```

is

```
int (char*)
```

and a pointer to such a function is of type

```
int (*)(char*)
```

Such pointers to "normal" functions are declared and used like this:

```
int f(char*);           // declare function
int (*pf)(char*) = &f; // declare and initialize pointer to function
int i = (*pf)("hello"); // call function through pointer
```

A similar syntax is introduced for pointers to members of a specific class. In a definition `mf` appears as

```
int S::mf(char*)
```

The type of `S::mf` is

```
int S:: (char*)
```

that is, "member of `S` that is a function taking a `char*` argument and returning an `int`." A pointer to such a function is of type

```
int (S::*)(char*)
```

That is, the notation for pointer to member of class `S` is `S::*`. We can now write:

```
// declare and initialize pointer to member function
int (S::*pmf)(char*) = &S::mf;

S a;
// call function through pointer for the object "a"
int i = (a.*pmf)("hello");
```

The syntax isn't exactly pretty, but neither is the C syntax it is modeled on.

A pointer to member function can also be called given a pointer to an object:

```
S* p;
// call function through pointer for the object "*p":
int i = (p->*pmf)("hello");
```

In this case, we might have to handle virtual functions:

```
struct B {
    virtual f();
};

struct D : B {
    f();
};

int ff(B* pb, int (B::*pbf)())
{
    return (pb->*pbf)();
};

void gg()
{
    D dd;
    int i = ff(&dd, &B::f);
}
```

This causes a call of `D::f()`. Naturally, the implementation involves a lookup in `dd`'s table of virtual functions exactly as a call to a virtual function that is identified by name rather than by a pointer. The overhead compared to a "normal function call" is the usual about 5 memory references (dependent on the machine architecture).

It is also possible to declare and use pointers to members that are not functions:

```
struct S {
    int a;
};

int S::* psm = &S::a;

void f(S* ps)
{
    ps->*psm = 2;
}

void g()
{
    S a;
    f(&a);
}
```

This is a complicated way of assigning 2 to a.a.

## 9 Overloading Sensitivity

The overloading mechanism can now distinguish between signed and unsigned values. For example:

```
overload f(int), f(unsigned);

void g1(int i, unsigned u)
{
    f(i);    // invoke f(int)
    f(u);    // invoke f(unsigned)
}
```

To support this the ANSI C notion of unsigned literals has been adopted:

```
void g2()
{
    f(1);    // invoke f(int)
    f(1u);   // invoke f(unsigned)
}
```

In the very near future the overloading mechanism will be able to distinguish between single and double precision floating point values. The long double floating point type specified in the draft ANSI C standard will also be adopted. For example:

```
overload f(float), f(double), f(long double);

void g3(float sp, double dp, long double ld)
{
    f(sp);    // invoke f(float)
    f(dp);    // invoke f(double)
    f(ld);    // invoke f(long double)
}
```

To support this the ANSI C notion of float and long double literals will have to be adopted:

```
void g4()
{
    f(1.);    // invoke f(double)
    f(1.f);   // invoke f(float)
    f(1.l);   // invoke f(long double)
    f(1);     // invoke f(double)
}
```

## 10 Resolutions

This section does not describe additions to C++ but gives answers to questions that have been asked often and do not appear to have clear enough answers in the reference manual. This section may be of most use to C++ compiler writers, but more adventurous users might also find it of interest.

### Function Argument Syntax

In argument declarations C++ chooses the longest type possible when there appears to be a choice:

```
typedef int I;
f(unsigned I);    // f takes an 'unsigned int' argument
g(const I);       // g takes an 'const int' argument
h(const I);       // h takes a 'const' argument (called 'i')
```

### Declaration and Expression Syntax

In processing declarations and expressions a C++ compiler looks at the complete declaration or expression to disambiguate:

```
T a;              // declaration: 'a' is a T
T b();            // declaration: 'b' is a function returning a T
```

```
T(*p1)(int);      // declaration: 'p1' is a pointer to a function
T(*p2)(7);         // expression: call 'p2' with the argument 7
```

Here, `p2` must be a previously declared pointer; `*p2` is cast to `T`; `T` must be a function or pointer to function type or a class type for which `operator()` has been defined.

```
T(*p3)();         // declaration: 'p3' is a pointer to a function
T(*p4)() = 7;      // expression: call 'p4' with no argument
```

For this to be correct, `p4` must be a previously declared pointer; `*p4` is cast to type `T`; `T` must be a function type, a pointer to function type, or a class type for which `operator()` has been defined. The value returned by `T(*p4)()` must be of a type such as `int&`.

```
T(*v)[7];         // declaration: 'v' is a pointer to vector
T(*u)[7]->m();     // expression: use the 7th element of T(*u)
```

Here, `u` must be a pointer to a previously declared vector; `*u` is cast to type `T`; `T` must be a pointer type or a class type for which `operator->` has been defined.

Redundant parentheses are illegal in declarations so token strings containing parentheses that would be redundant in a declaration are interpreted as expressions:

```
T a;              // declaration: 'a' is a T
T (b);            // expression: cast 'b' to T

T c();            // declaration: 'c' is a function returning a T
T (d)();          // expression: cast 'd' to T
```

For the last expression to be valid, `T` must be a function type, a pointer to function type, or a class type for which `operator()` has been defined.

### Scope of Enumerators

An enumerator is entered in the scope in which the enumeration is defined. For example:

```

void f() {
    ...
    enum { a, b };
    int x = a;    // ok: 'a' is in scope
    {
        int y = a;    // ok: 'a' is in scope
        ...
    }
    ...
    int z = a;    // error: 'a' is not in scope
    ...
}

```

In this context a class is considered a scope and the usual access control rules apply. For example:

```

class X {
    enum { x, y, z };
    ...
public:
    enum { a, b, c };
    f(int i = a) { g(i+x); ... }
    ...
}

void h() {
    int i = a;    // error: 'X::a' is not in scope
    i = X::a;    // ok
    i = X::x;    // error: 'X::x' is private
}

```

## Function Types

It is possible to define function types that can be used exactly like other types, except that variables of function types cannot be defined only pointer to function variables:

```

typedef int F(char*); // function taking a char* argument
                        // and returning an int
F* pf;                // pointer to such function
F f;                  // error: no variables of function type allowed

```

Function types can be useful in friend declarations; Here is an example from the C++ task system:

```

class task : public scheduler {
    friend SIG_FUNC_TYP sig_func; // the type of a function must be specified
                                // in a friend function declaration
    ...
}

```

The reason to use a typedef in the friend declaration sig\_func and not simply to write the type directly is that the type of signal() is system dependent:

```

// BSD signal.h:
typedef void SIG_FUNC_TYP(int, int, sigcontext*);

// 9th edition signal.h:
typedef void SIG_FUNC_TYP(int);

```

Using the typedef allows the system dependencies to be localized where they belong: in the

header files defining the system interface.

### Anachronisms

A C++ compiler is not obliged to implement the compatibility hacks mentioned in sections 15.2 and 15.3 of the C++ reference manual. The following examples are *not* C++:

```
int f(a) char* p; { ... }; // old style function definition

struct s { ... };
int s ();                // two declarations of 's'

int c.f() { ... };      // dot as membership operator, use ::
```

It would be unwise for a user to rely on these features.

To ease use of common C++ and ANSI C header files void may still be used to indicate that a function takes no arguments:

```
extern int f(void);      // same as "extern int f();"
```

### 11 Conclusions

C++ is holding up nicely under the strain of large scale use in a diverse range of application areas. The extensions added so far have been relatively easy to integrate into the C++ type system. The C syntax, especially the C declarator syntax, has consistently caused much greater problems than the C semantics; it remains barely manageable. The stringent requirements of compatibility and maintenance of the usual run-time and space efficiencies did not constrain the design of the new features noticeably. Except for the introduction of the keywords *private* and *protected* the extensions described here are upward compatible.

### 12 Acknowledgments

Most of the credit for these extensions goes to the literally hundreds of C++ users who provided me with bugs, mistakes, suggestions, and most importantly with sample problems.

Phil Brown, Tom Cargill, Jim Coplien, Steve Dewhurst, Keith Gorlen, Laura Eaves, Bob Kelley, Brian Kernighan, Andy Koenig, Stan Lippman, Larry Mayka, Doug McIlroy, Pat Philip, Dave Prosser, Roger Scott, Jonathan Shopiro, and Kathy Stark supplied many valuable suggestions and questions.

The C++ multiple inheritance mechanism was partially inspired by the work of Stein Krogdahl from the University of Oslo<sup>3</sup>.

### 13 References

- [1] Tom Cargill: *PI: A Case Study in Object-Oriented Programming*. OOPSLA'86 Proceedings, pp 350-360, September 1986.
- [2] Gul, Agha: *An Overview of Actor languages*. SIGPLAN Notices, pp 58-67, October 1986.
- [3] Krogdahl, Stein: *An Efficient Implementation of Simula Classes with Multiple Prefixing*. Research Report No. 83 June 1984, University of Oslo, Institute of Informatics.
- [4] Snyder, Alan: *Encapsulation and Inheritance in Object-Oriented Programming Languages*. SIGPLAN Notices, November 1986, pp 38-45.

- [5] Stroustrup, Bjarne: *The C++ Programming Language*. Addison-Wesley, 1986.
- [6] Stroustrup, Bjarne: *Multiple Inheritance for C++* Proc. EUUG Spring'87 Conference. Helsinki.
- [7] Stroustrup, Bjarne: *What is Object-Oriented Programming?* Proc. 1st European Conference on Object-Oriented Programming. Paris, 1987. Springer Verlag Lecture Notes in Computer Science, Vol 276, pp 51-70.
- [8] Stroustrup, Bjarne: *Possible Directions for C++* Companion paper.
- [9] *Lisp Machine Manual*. Symbolics, Inc. 1981.

1. The first part of the paper discusses the  
 2. background and motivation for the work.  
 3. The second part describes the system architecture.  
 4. The third part presents the experimental results.  
 5. The fourth part discusses the conclusions and future work.  
 6. The fifth part is the bibliography.  
 7. The sixth part is the appendix.  
 8. The seventh part is the index.  
 9. The eighth part is the list of figures.  
 10. The ninth part is the list of tables.

# Two extensions to C++: A dynamic link editor and inner data

Philippe Gautron

Marc Shapiro

Institut National de Recherche en Informatique et Automatique

B.P. 105, 78153 Le Chesnay Cedex, France

uucp: gautron@corto.inria.fr

## Abstract

We present the design and implementation of a dynamic loader and linker for the C++ language. The motivations and goals of this tool are discussed, as well as its introduction into a C++ compilation chain. We also discuss "inner data", an alternative style of single inheritance, and a complement to dynamic linking. In each case, a keyword was added to the C++ language, and a few modifications to the compiler were necessary. The implementation is clean and portable. We present the rationale of this work, the necessary changes to the language, and a working implementation.

## 1 Introduction

The work described here is part of SOS, a distributed object-oriented operating system for an integrated office-automation environment [Sha86]. In SOS, objects are routinely migrated from one machine and/or one address space to another. The code for the object is migrated along with its data. Thus the code of the object must be dynamically linked into the receiving address space. This is accomplished by making invocations be indirect procedure calls, via a per-instance table which points to the procedures. Our system detects when a piece of code is needed, loads it, and fills the corresponding procedure table. The first part of this article describes the design and implementation of the dynamic linker, as well as its integration in the C++ environment. The compiler manages the tables and extra indirections, and generates the code to perform detection, loading and linking.

The data for an object has a system part (a descriptor) and a programmer-defined part. We wish

to decouple the two parts, to allow flexibility in user-part management, and to protect the system part. However, we want the user of an object to see only one object, and to describe the relation between the parts by inheritance. The second part of this article relates our experiment with "inner data", an alternative style of single inheritance. Data for a class derived by inner is allocated in a separate block and accessed by an extra indirection. The extra indirections are automatically generated by the compiler.

A full description of SOS is outside of the scope of this paper, which is restricted to the description of our modified C++ environment on Unix 4.2BSD.

## 2 Dynamic Linking

The conventional production cycle for a C or C++ program is a sequence of edit-compile-link-load-run phases.

Full linkage before execution, or static linking, is often imposed by the programming environment, and is not always desirable. Dynamic linking, i.e. delaying the binding between procedure names and their code until execution time, yields a number of benefits:

- Program images are smaller. Procedures are read at run time as needed, and do not take up redundant space image on disk.
- The production cycle time is shorter. Static linking with a large library can take minutes.
- Shared libraries can be more easily implemented. A dynamic loader can recognize at run-time a library which is already in use, and arrange to share its in-core image.
- Greater flexibility. With dynamic linking, each execution is guaranteed to use the most recent version of all the procedures.

\* This research is financed in part by the European Community, under Esprit contract SOMIW no. 367.

The main advantage of static over dynamic linking is reduced execution time: a dynamically-linked program incurs the overhead of linking at each execution, whereas static linking is done only once. For this reason, it is desirable to allow a single program to combine both static or dynamic linking, as necessary.

Dynamic linking may be added to compiled languages, if there is sufficient information in the binary file to: (1) detect that a new procedure needs to be imported; (2) extract the symbolic name (and possibly, type information) of that procedure; (3) discover the file containing the procedure's code.

In the Multics operating system [Org72], the program loader performs dynamic linking for any program in any language.

An attractive alternative is an implementation which does not impact the kernel.<sup>1</sup> For instance, the Andrew project uses "Camphor," a dynamic linking system for C programs. Camphor consists of a set of preprocessor macros, support programs, and run-time libraries (linked statically).

CLAM [CCM87] is an object-oriented system for graphical user interfaces based on the client/server model. CLAM allows dynamic loading of C++ classes into the server, using a special class which manages the loading of dynamic classes. A library function must be statically linked with each client to establish connections. CLAM does not support shared libraries, uses a modified version of the standard loader `ld`, is BSD-format-dependent, but integrates a stub generator to interface with dynamic linking.

We present here a dynamic linker for the C++<sup>2</sup> object-oriented programming language [Str85]. Our work integrates linker support into the compiler. This requires a small addition to the syntax of C++, and some additions to its code generator. We link the code for a class at the time of its first instantiation.

Our implementation builds upon Camphor. Like Camphor, we keep a static link phase, and do not support shared libraries. Our implementation is clean and machine-independent. It depends only on the format of binary files (both `a.out` and COFF versions are available).

This paper distinguishes between the *exported* relocatable modules, and the *importer*, i.e. the program into which these modules are linked at run time.<sup>3</sup>

<sup>1</sup>Although an implementation outside the kernel cannot efficiently support shared libraries.

<sup>2</sup>Currently C++ version 1.1, with known bugs fixed.

<sup>3</sup>This distinction is made essentially for the sake of clarity of exposition, as rôles shift during execution. After linkage, an exported module becomes part of the importer, and its execution may cause further importations.

The use of our dynamic linking system may be divided into four independent steps:

- compile-time *definition* of exported modules by the exporter
- compile-time *declaration* of imported modules by the importer
- run-time *detection* that the importer needs to load a new module, by the linking system
- run-time *loading* and *link-editing* the module into the importer.

The detection step is performed at run-time by code generated by the compiler. The last step is performed by a run-time support library.<sup>4</sup>

## 2.1 Dynamic linking in the C++ compiler

Dynamic modules are intimately bound to the class concept.

The new keyword `dynamic` may be added to a class declaration. For a dynamic class, the code of methods is either imported (for declarations) or exported (for definitions). The code generated for calls to member functions of a dynamic class uses an indirection through a table of procedure pointers.<sup>5</sup> The compiler generates code to detect when the methods are actually needed. Then a (statically-linked) procedure will be called at run time, to load in its code, resolve addresses, and fill the table with legal values.

We will present, both our design, and the actual modifications to the generated code.

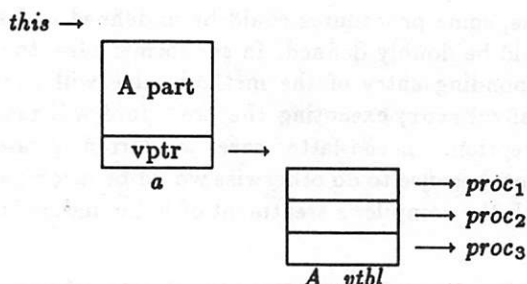
### 2.1.1 Declaration, detection and invocation of a dynamic class

In an usual C++ class, non-virtual methods are called directly by name, whereas virtual methods are called indirectly via a *method table*. Virtual constructors are not allowed. This table (one per class) is allocated in static memory. For a class `A`, it is named `A_vtbl`. Its address is stored in a supplementary field (one per instance), named `vptr`. Binding, i.e. filling `vptr` with the address of the table, is done within the execution of the constructor.

The following example shows the layout of an instance `a` of a regular (non-dynamic) class `A`, with virtual methods, generated by (both versions of) the compiler:

<sup>4</sup>Which is linked statically into the importer.

<sup>5</sup>This is similar to the treatment of virtual methods. Inline declarations are ignored for procedures of a dynamic class.



A dynamic class is a class for which *all* methods are called indirectly.<sup>6</sup> One declares a dynamic class A by adding the keyword `dynamic` in front of the class declaration:

```

dynamic class A {
    int a;           // field
    int ma();        // simple method
    virtual int va(); // virtual method
    A (args);        // constructor
};
  
```

In the importer, loading the code for class A is delayed until it is first needed. This always occurs when executing a constructor. In addition, initialization of the method table must take place before the constructor is called. The modified compiler generates detection code for every instantiation of a dynamic class:

```

A a1 (args);
A* a2 = new A (args);
  
```

will be generated into:<sup>7</sup>

```

int (** A__vtbl)();
struct A a1;
struct A* a2;

A__vtbl = sosFindCode ("A", 0);
// import code of class A
// 0 ends the argument list
(* A__vtbl[ 2 ]) (& a1, args);
// call the constructor
// assuming 2 is its index

A__vtbl = sosFindCode ("A", 0);
a2 = (A*) (* A__vtbl[ 2 ]) (0, args);
  
```

The procedure `sosFindCode` is statically linked with the importer, and calls the importation mechanism if necessary, i.e. if the code has not already been loaded.

Member functions of a dynamic class behave similarly to virtual methods. So, as before:

<sup>6</sup>We keep the names `vptr` and `vtbl` for practical reasons.

<sup>7</sup>Compare with the code generated by the unmodified compiler, in Appendix A.

- The `vptr` field of the instance is filled by the constructor.
- Indexes into the table are computed at compile-time.
- The first argument to the constructor is either the address of the new instance, or 0 if space is to be allocated.

The name of the imported file may be explicitly given by optional arguments to the dynamic declaration, as in:

```

dynamic ("export.o") class A {
    // declarations
};
  
```

Such filenames are passed as extra arguments to the `sosFindCode` call; e.g. for the above declaration:

```

A__vtbl = sosFindCode ("A", "export.o", 0);
  
```

## 2.1.2 Implementation of dynamic load and linking

The work of the dynamic loader/linker is twofold: first, the exported code is read, and its relocatable data is updated. Second, the importer's method tables are filled with the actual location of the text.

### Preparation of the exported code

A special preparation of the exporter is necessary. An additional compilation option and a post-processing step (similar to the "patch" C++ post-processor<sup>8</sup>) produce an exportable module. Program `makezport` creates this module:

```

makezport -o export.o f1.c f2.c ...
  
```

Here, `f1.c`, `f2.c`, ... are compiled and bound into a single exportable binary file `export.o`.

A distinguished procedure per source file is generated, called `_ZINIT_f1.c_` for a file named `f1.c`. When called after relocation, it will return the resolved method tables, for all dynamic classes of the file. If a method is not defined in this file, the corresponding pointer is set to `sosDynError`.

Compilation also includes the following structure `_Zlink` into each file:

<sup>8</sup>The C++ post-processor is used to handle the constructors and destructors of static instances. Two versions are provided: "munch" for BSD, and "patch" for System V. We have ported "patch" to BSD (it is more efficient than "munch"), adding the preparation phase explained here.

```
static struct __Zlink{
    struct __Zlink* next; // next file
    int (*ctor)(); // static constructors
    int (*dtor)(); // static destructors
    int* (*export)(); // export procedure
} _ZLINK_f1_c = { // for file f1.c
    0, 0, 0, _ZINIT_f1_c
};
```

The first field, `next`, is used by the post-processor to chain such structures together. The second, `ctor`, points to the procedure which will call static constructors for the current file. The third, `dtor`, is similar for static destructors. The fourth is the distinguished procedure which returns the method tables for this file.

`Makeexport` matches all symbols beginning with the prefix `_ZLINK_`, chains them via the `next` pointer, and makes the "entry point" of the binary file `export.o` point to head of this list. This "entry point" is a standard entry of any binary file, and is read by the dynamic linker when importing `export.o`.

## Dynamic Load

A call to `sosFindCode` is inserted by the compiler before each instantiation of a dynamic class. It is the responsibility of `sosFindCode` to:

- keep track if the code for a given class is already loaded or not.
- if not already loaded, find the corresponding binary file, load and relocate it, and bind unresolved references.
- fill in the method table with resolved references. This table is allocated in the static data, imported with the module, and does not require supplementary allocation.

The call to `sosFindCode` causes the exported binary to be loaded into the importer's data area.

Note that the above is different from the strategy used in Camphor and similar implementations, where dynamic linking is a side-effect of the first call to a dynamic procedure. Our strategy (linking at instantiation time) incurs some overhead, but it is more portable and more extensible. We could change to the Camphor strategy by making `sosFindCode` a null procedure, and performing the above algorithm in `sosDynError` instead.

The addresses of the methods defined in `export.o` are known by walking through the list from the "entry point", executing all the procedures described above.

As in standard C++, one may define more than one class per source file, or spread a class definition

over many source files. Therefore, at dynamic link time, some procedures could be undefined and some could be doubly defined. In the former case, the corresponding entry of the method table will point to `sosDynError`; executing the procedure will raise an exception. In the latter case, we currently raise an error, because to do otherwise would be incompatible with the compiler's treatment of inline methods.

## 2.1.3 Importer and external procedures

Calls to importer and external procedures from within the exporter must also be linked dynamically. We choose not to generate indirect calls in this case, but to keep direct calls.

For all importer files, our modified C++ generates a procedure which handles a table of triples (name, type, address), for all externals defined in this file, where type is either 1 for procedure or 0 for data. This procedure, called `_ZLINK_importer_c_` for a file named `importer.c` will call an initialization procedure `_Ztable_from_cfront` of the additional library. It is added to the front of the list of static constructors and is automatically called before execution starts.

Here is an example for file `importer.c`:

```
static int _ZLINK_importer_c_(){
    static struct{
        char* n; char t; int (*ptr)();
    } tbl[] = {
        "_aFunction", 1, aFunction,
        // aFunction defined in importer.c
        "_aDatum", 0, aDatum,
        ...
        0, 0, 0
    };

    _Ztable_from_cfront (tbl);
    // initialize the dynamic linker with tbl
}
```

All external procedures possibly called by imported code must be statically linked with the importer, and described by a similar table. For example, the description of the C library contains:

```
struct library libc[] = {
    "_printf", printf,
    ...
};
```

When a exported module is loaded, the dynamic linker resolves references to importer and external procedures by searching through the above-described tables.

### 2.1.4 Dynamic instances

An instance of a dynamic class may itself be declared dynamic. A dynamic instance imports not only its code, but also its data. Consider dynamic class A:

```
dynamic ("export1.o") class A {
    // declarations
};

f(){
    A* a1 = new A (args);
        // non-dynamic instance
    A* a2 = new dynamic ("export2.d") A (args);
        // dynamic instance
}
```

For the first instantiation, a call to `sosFindCode` is generated. For the second one, the compiler generates a call to `sosImport`, before the call to the constructor. Either procedure can call the code-loading mechanism if necessary.

The filename where the data are found, `export2.d`, is given to the `sosImport` procedure, which returns a pointer to the imported data.

```
int (** A__vtbl) ();

f(){
    A* a1;
    A* a2;

    A__vtbl = sosFindCode ("A", "export1.o", 0);
    a1 = (A*) ( * A__vtbl[ 2 ] ) (0, args);

    a2 = (A*) sosImport (& A__vtbl, "A",
        "export2.d", "export1.o", 0);
    a2 = (A*) ( * A__vtbl[ 2 ] ) (a2, args);
}
```

Note that the first argument of the constructor call for `a2` is no longer 0, as in previous examples, but the instance pointer, so the usual allocator mechanism in the constructor is not affected.

This scheme requires restrictions on dynamic instantiation. Dynamic instances cannot be static or stack-allocated.

In SOS [SAHM87], the dynamic instance declaration designates, not a file name, but the name of a service, that may create the data on-the-fly. For instance, in:

```
A* a = new dynamic ("service_A") A (args);
```

the object manager searches for a server for `service_A`, which is asked to create the data for object `a`. The service attaches a method table, pointing to some code, to `a`. The data and, if necessary, the code

are migrated together into the importer. Thus, the code for object `a` may be different than the code used by another instance of class `A`.

### 2.1.5 Inheritance

In the current implementation, a dynamic class can derive from any class, but a non-dynamic class cannot derive from a dynamic one.

The constructor of the base class is called in the body of the constructor, preceded by a call to `sosFindCode` for the base class. This call allows use of the suitable method table in the body of the base constructor. Note that this mechanism is similar to the assignment of the virtual pointer of the base class in the unmodified C++ (cf. Appendix A).

### 2.1.6 Concurrency

If parallelism with shared memory is possible, or if one wishes to allow several instances of a same dynamic class with different method tables, the static `vp_ptr` pointer is a bad solution. Generally, global variables should be avoided.

The above examples of generated code were simplified. In fact, we add an extra temporary argument to all constructors of a dynamic class. Consider again the dynamic class `A`:

```
f(){
    A* a1 = new A (args);
    A* a2 = new dynamic ("service_A") A (args);
}
```

The following code is generated:

```
f(){
    A* a1;
    A* a2;
    int (** tmp)();

    tmp = sosFindCode ("A", "export.o", 0);
    a1 = (A*) ( * tmp[ 2 ] ) (0, tmp, args);

    a2 = (A*) sosImport (& tmp, "A",
        "export2.d", "service_A", 0);
    a2 = (A*) ( * tmp[ 2 ] ) (a2, tmp, args);
}
```

Note the second extra argument `tmp` to the constructor call.

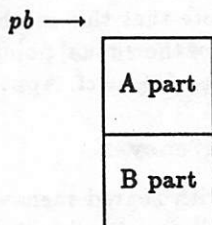
The generated code of the constructor is thus completed, instead of affecting `vp_ptr` with the global variable `A__vtbl`:

```
A_ctor( A* this, int (** vtbl)(), args ){
    ....
    this->vp_ptr = vtbl;
    ....
}
```

### 3 Inner data

Inheritance, in the unmodified compiler, is the ability of a class to add new fields to a base class. Allocation of instances is in a monolithic block, as follows:

```
class A {
    // A data
};
class B : A {
    // B data
};
B* pb = new B;
```



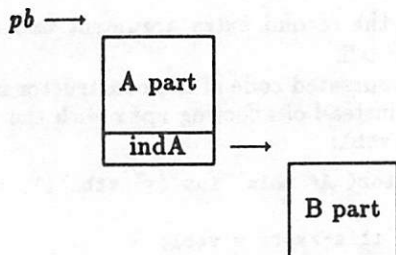
In an alternative style of inheritance, inner data is the ability for a derived class to add new fields, via a pointer field in the base class. The keyword `inner` is used, both to localize the pointer in the base class declaration, and to declare an "inner inheritance". Consider the following declaration:

```
class A {
public:
    int a;
    inner indA; // interpreted as: void* indA;
    void ma();
};

class B : inner public A {
public:
    int b;
    void mb();
};

B* pb = new B;
```

This corresponds to the following data allocation:



The inner declaration is interpreted as an untyped pointer. The usual inheritance mechanisms (visibility rules, method access and virtual methods) are not affected by inner declarations. The only modifications are memory allocation, and an extra indirection in accessing the B part of instances of class B. This indirection (and the necessary temporary variables) are automatically generated by the compiler. Thus the above instantiation of class B will be generated into:

```
A* tmpA;
B* tmpB;

pb = ( tmpA = new A, tmpB = new B,
        tmpA->indA = tmpB, tmpA );
```

The following assignments:

```
pb->a = 0;
pb->b = 0;
```

will be generated into:

```
A* tmpA;
B* tmpB;

tmpA = (A*) pb, tmpA->a = 0;
tmpB = (B*) (((A*) pb)->indA), tmpB->b = 0;
```

Method calls do not require any extra indirection. The first argument is, as usual, the pointer to the current instance (cf. Appendix A). Extra indirections will be generated in the *body* of the method.

With inner data, there is more flexibility in the localization of the derived part. Derived parts can be moved just by assignment of the inner pointer:

```
B* b1 = new B; // A1 and B1 parts handled by b1
B* b2 = new B; // A2 and B2 parts handled by b2

b1->indA = b2->indA;
// A1 et B2 parts handled by b1
```

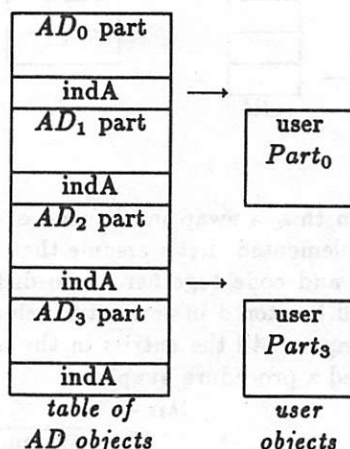
The allocation of static and stack-based objects are not under user control. Therefore, inner-derived objects must be instantiated by `new` and freed by `delete`.<sup>9</sup>

#### 3.1 Example

In SOS, all user classes are derived from a base class `sosObject`, which is itself derived from kernel class

<sup>9</sup>Note that this limitation is the same as in Minimal C++, an Apple port of C++, which requires a special memory handler. Such storage limitations are also required in other extensions of C++.

AD.<sup>10</sup> The `sosObject` class has no data and only represents the interface between kernel and user objects. Instances of class AD are allocated in a table in protected memory:



The inner inheritance allows both, to keep descriptors of constant size in protected memory, and, for the kernel, to manage references on user object parts. The size of an user object cannot be known in advance, so usual inheritance would not allow to easily manage objects in protected memory. With the inner mechanism, the indirection through the AD is not seen by the users. Both the users and the system refer to an object by the address of its AD.

Furthermore, the inner inheritance is masked to users, who only see the usual inheritance, by class `sosObject`. This is illustrated below in class `userObject`:

```
// kernel declaration
class AD {
    // AD declaration
    inner indAD;
};

class sosObject : inner AD {
    // interface declaration
};

// user declaration
class userObject : sosObject {
    // members of userObject
};
```

### 3.2 Sizeof

`Sizeof` is a C++ operator treated by the translator. If class B is derived by inner from A, should `sizeof`

<sup>10</sup>Acquaintance Descriptor. This class handles contains the kernel's information about objects (such as its Object Identifier, etc.).

(B) return just the size of the B part, or the total A+B? In our implementation, the result of `sizeof` is identical to usual inheritance, i.e. `sizeof (B)` returns the sum. A new operator, `innersizeof`, returns only the size of the inner part:

```
B* b = new B;
int i = sizeof( *b ); // A part + B part
int j = innersizeof( *b ); // B part only
```

### 3.3 Using both styles of inheritance

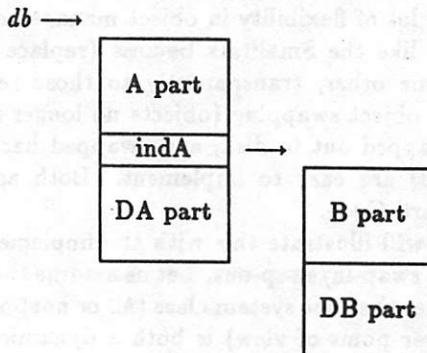
Both styles of inheritance can be used together. For instance:

```
class A {
    int a;
    inner indA;
};

class DA: A { int da; };
class B: inner DA { int b; };
class DB: B { int db; };
```

`DB* db = new DB;`

The data of `db` is laid out thus:



### 3.4 Cost of inner data

The cost of inner data is the allocation of temporary variables, extra memory allocations and access through indirection.

Temporary variables are generated as needed by each statement but are reused from one statement to another. A separate memory allocation is generated for each inner part. One additional assignment, for the base part, plus one indirection, for the inner part, are needed to access fields. Let us consider the following example:

```
class A {
    int a;
    inner indA;
};
```

```
class B : inner A { int b; };
```

```
B* pb = new B;
pb->a = 1;
pb->b = 2;
```

The generated code for the last three instructions will be:

```
A* tmpA;
B* tmpB;
B* pb;
```

```
pb = ( tmpA = new A, tmpB = new B,
        tmpA->indA = tmpB, tmpA );
tmpA = (A*) pb, tmpA->a = 1;
tmpB = (B*) (((A*) pb)->indA), tmpB->b = 2;
```

Two extra variables are thus allocated and the instruction cost, in terms of assignments, is a factor of two for one level of inner indirection.

## 4 An example: object swapping

The use of dynamic linking along with inner data allows a lot of flexibility in object management. Operations like the Smalltalk become (replace an object by some other, transparently to those referring to it), or object swapping (objects no longer in use can be swapped out to disk, and swapped back in when needed) are easy to implement. Both are hard in standard C++.

We will illustrate this with the implementation of object swap-in/swap-out. Let us assume the following scheme: the root system class (AD or `sosObject` from the user point of view) is both a dynamic class and contains an inner declaration:

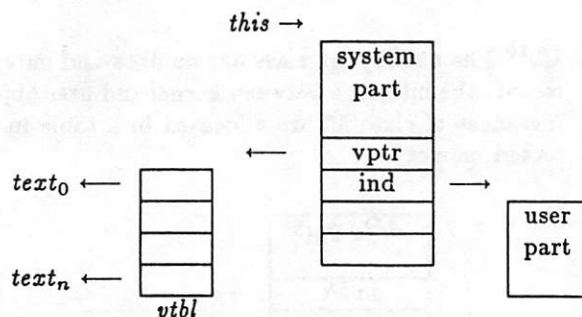
```
class AD {
    // system part
    inner ind;
};
```

```
dynamic class sosObject : inner AD { ... };
```

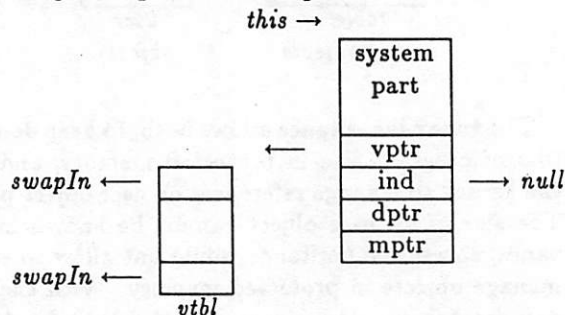
A user class inherits from this by usual inheritance:

```
dynamic class userObject: sosObject {
    // user part
};
```

The kernel part of a `userObject` instance contains two pointers, one for the dynamic methods (`vptr`), the other for the inner data (`ind`):



Based on this, a swap mechanism of user objects can be implemented. Let's assume that we swap out both data and code together on to disk. Their locations will be stored in two extra fields of the AD, `dptr` and `mptr`. All the entries in the method table are assigned a procedure `swapIn`:



When any method is called, it actually executes `swapIn`. This routine loads the data and code from disk, and restores `ind` and `vptr` to significant values. The called method is then restarted.

## 5 Conclusion: the advantages of the object-oriented approach

Our design is well-integrated with the concepts of the object-oriented approach.

### Dynamic classes

- encapsulation:  
The unit of binding or importation is the class or the instance. C++ performs the necessary compile-time checks, without clumsy macro-generation.
- instantiation:  
Importation is performed at a well-known time, instantiation. This approach is an

advantage in the context of an open operating system where users must control system mechanisms.

- **dynamic binding to code:**  
Each instance has its own method table, enforced at run-time. A table can be changed it at any moment of run-time, providing flexibility.

#### Inner data

- **inheritance:**  
Inner data adds flexibility to the usual inheritance mechanism, without modification of access from the programmer's point of view. The run-time overhead is increased, however.
- **dynamic binding to data:**  
The data inside an object may be moved by a simple assignment, transparently to the current users of the object. This allows simple implementations of Smalltalk's become, or of object swapping.

C++ provides the efficiency necessary for an operating system implementation. Benchmarks on the different uses of class methods (simple, inline or virtual) and communicated by B. Stroustrup showed that the overhead of indirect calls to virtual methods is negligible. Our implementation generalizes indirect calls to all methods of a dynamic class.

A first implementation currently runs on Sun-3 and Vax running BSD Unix. It is portable as long as the C++ compiler is. The only dependency is the format of the loader, BSD or COFF.

The current version does no name or type-checking. There is no real assurance that the imported file is the right one. The importer's method tables are assigned what is returned by the imported entry procedure, without any checking of number or sizes or contents. Clearly this is undesirable. Our next developments will be to implement a full name and type-checking at importation time, with minimal overhead.

## Acknowledgments

Our acknowledgments to authors of Camphor: P. Smith, B. Sacks, S. Daniel, M. Kazar, D. Nichols, A. Palay and F. Hansen; to the C++ author, B. Stroustrup; to Y. Gourhant for port of dynamic linker to COFF; to our colleagues of the SOMIW project: V. Abrossimov, S. Habert, M. Makpangou, L. Mosseri; and to R. Ehrlich for his invaluable help.

## Appendix A: Code generated by the unmodified translator

This appendix is provided to compare the code generated by the unmodified translator with the code generated by our translator. First, let us consider the following declarations of two classes, a base class A and a derived class B, their constructors and a procedure f containing two instantiations.

```
// base class
class A {
    int a;                // field
public:
    A (int);              // constructor
    void m1 (int);        // simple method
    virtual m2 (int);     // virtual method
    virtual m3 (int);     // virtual method
};

// derived class
class B : public A {
    int b;
public:
    B (int);
};

// constructor definitions
A::A (int i) { a = i; }
B::B (int i) : (i) { b = i; }

// procedure f
void f(){
    A a (100);            // allocated on stack
    B* b = new B (100);   // allocated in free store

    a . m1 (100);
    b -> m3 (100);
}
```

The generated code will be, simplified:

- for the class declarations:

```
struct A {
    int _A_a ;
    int (**_A_vptr )();
};

static int (* _A_vtbl[])( ) = {
    (int(*)()) _A_m2,
    (int(*)()) _A_m3, 0
};

struct B {
    int _A_a ;
    int (**_A_vptr )();
};
```

```

    int _B_b ;
};

static int (* B_vtbl[])() = {
    (int(*)()) _A_m3,
    (int(*)()) _A_m2, 0
};

```

Note the supplementary field `_A_vptr` in the class declarations and the two extra declarations `A_vtbl` and `B_vtbl`. These pointers are needed for the virtual procedure calls.

- for the constructors:

```

struct A*
_A_ctor (this ,i )
    struct A* this; int i;
{
    if (this==0)
        this = (struct A*)_new (8) ;
    this->_A_vptr = A_vtbl;

    this->_A_a = i ;
    return this ;
}

struct B*
_B_ctor (this ,i )
    struct A* this; int i;
{
    if (this==0)
        this = (struct B*)_new (12) ;
    this = (struct B *)
        _A_ctor((struct A*)this, i);
    this->_A_vptr = B_vtbl;

    this->_B_b = i ;
    return this ;
}

```

Data allocation is handled by `_new`, with a size argument, which calls the `malloc`. Note the call to the A constructor within the body of the B constructor, and the assignment of the virtual pointers.

- For procedure f:

```

int f (){
    struct A* a;
    struct B b;

    a = (struct A *)
        _A_ctor ((struct A *)0, 100);
    _B_ctor (& b, 100);

    _A_m1 (& a, 100);
    (* (b->_A_vptr[1]) )

```

```

    ((struct A *)& b, 100);
}

```

Note the extra first argument to constructors and methods, and the index of the virtual method.

## References

- [CCM87] L.A. Call, D.L. Cohrs, and B.P. Miller. Clam — an open system for graphical user interfaces. In *OOPSLA'87*, Orlando, Florida (USA), October 1987.
- [MSC\*86] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosental, and F. D. Smith. Andrew: a distributed personal computing environment. *Communications of the ACM*, 29(3):184-201, March 1986.
- [Org72] E.I. Organick. *The Multics system: an examination of its structure*. MIT Press, Cambridge, Mass. (USA), 1972.
- [SAHM87] Marc Shapiro, Vadim Abrossimov, Sabine Habert, and Mesaac Mounchili Makpangou. *Un recueil de papiers sur le système d'exploitation réparti à objets SOS*. Rapport Technique INRIA 84, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), May 1987.
- [Sha86] Marc Shapiro. SOS: a distributed object-oriented operating system. In *2nd ACM SIGOPS European Workshop, on "Making Distributed Systems Work"*, Amsterdam (the Netherlands), September 1986. (Position paper).
- [Str85] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1985.





# The Architecture of a C++ Compiler

S. C. Dewhurst

AT&T

## ABSTRACT

This paper discusses several aspects of project and design goals applied to development of a C++ compiler written in C++ and the results obtained. The work described was undertaken at AT&T jointly by Laura Eaves, Kathy Stark and the author. The intent is not to present a detailed description of the compiler internals, but rather to lay out in a general way how we think about issues central to the design of a compiler, and how these perceptions have affected the compiler design.

## Introduction

The first section describes some of the considerations that have shaped this compiler design. These include considerations for efficiency and portability, compatibility with the existing translator, code and perceived language standards, corporate product goals, and the developers' *own* goals.

An overview of the structure of the compiler shows how the design reflects these goals, and serves as background for the remaining sections. This design shows clearly how C++ supports the effective melding of multiple programming paradigms: Subsections of the compiler are reflective of procedural, data abstraction, and object-oriented modes of programming, as well as hybrids of these.

The bulk of the paper addresses two important properties of the compiler: Its extreme portability and its ability to function within multiple compilation paradigms without sacrificing compile-time performance.

A final section offers gratuitous advice in presenting a number of general principles of compiler design gleaned from this project.

## Aspects of the Design Process

Compiler design has been described as "the Art of Compromise",<sup>[1]</sup> in that there are competing demands placed on the compiler writer to make the compiler and/or generated code small and/or fast and/or to make the compiler easily retargetable, and/or cater to sophisticated and/or naive users. It is an unfortunate fact that, in the language of users and management, conditional expressions like the above do not generally have a well-defined evaluation order and rarely allow for short-circuiting. It is also often the case that, as a project progresses, market conditions may change or elements of reality may enter the management decision-making process and cause the original goals and constraints to be modified.

The C++ compiler project had additional complicating factors. The most fundamental problem was the result of C++'s being a new language: The detailed semantics (and syntax) were not defined in a uniform and formal way. Throughout the beginning stages of the project, the C++ language definition was jointly and variously resident in *The C++ Programming Language*,<sup>[2]</sup> the existing C++ Translator (cfront), and user code. Bringing these three together into a coherent language definition that

could be implemented was further complicated by C++'s continued evolution throughout this process, adding major features like multiple inheritance. An additional goal is that C++ should have no "superfluous" differences with the emerging ANSI C standard,<sup>[3]</sup> effectively merging two dynamic language standards with a highly qualitative condition for conflict arbitration.

Finally, as is always is the case (but rarely admitted), we developers had our *own* goals for the project: The compiler should be portable in a very general sense, with as few dependencies as possible on machine, operating system, or mode of use as possible.

The compiler was designed for portability from the start, but we added other design goals later. As the project progressed, we realized that the compiler should be able to function as a general utility in a variety of programming environment configurations. We arrived at this realization for three major reasons: First, we expect one major use of C++ will be to produce applications very quickly by using inheritance to patch together and customize libraries of classes using, for a given application, only a small percentage of the code and interface of each library. Clearly as the size of applications and number of libraries available increases, it will be necessary to avoid useless recompilation as much as possible. Second, tools in a programming environment have a difficult time obtaining information about C++ programs. Because extraction of non-trivial information typically involves some level of compilation, the compiler should be able to serve as a general utility to provide information about C++ programs to other programming tools. Third, techniques developed to support C++'s scope semantics (rather unexpectedly) caused us to view the process of compilation from a new perspective, which in turn allowed us to redesign the compiler for use under a variety of models of compilation.

## Architecture Overview

The ensemble of goals and constraints listed above for the compiler imply that the compiler architecture should have two general properties: First, it should be designed for redesign. This is an overriding concern when neither the language to be compiled nor the compilation paradigm is fixed. Second, it should be amenable to the application of heuristic as experience is gained in how C++ is used. After the gross structure of a compiler is determined, most of the task of producing a "quality" implementation in the senses of compile and run time performance is simply engineering; the design should allow the implementors the ability to experiment with a variety of data structures and algorithms in specific domains without affecting the rest of the compiler. Finally, aspects of the portable structure of the compiler require that certain modules be modifiable or replaceable without affecting the rest of the compiler.

This list of general requirements isn't very different from those of many software design projects. It is interesting to note that these are essentially the problems addressed by data abstraction, object-oriented programming and hacking; three paradigms C++ supports well. Throughout the design and implementation of the compiler we made use of these paradigms, both in their "pure" and modified forms.

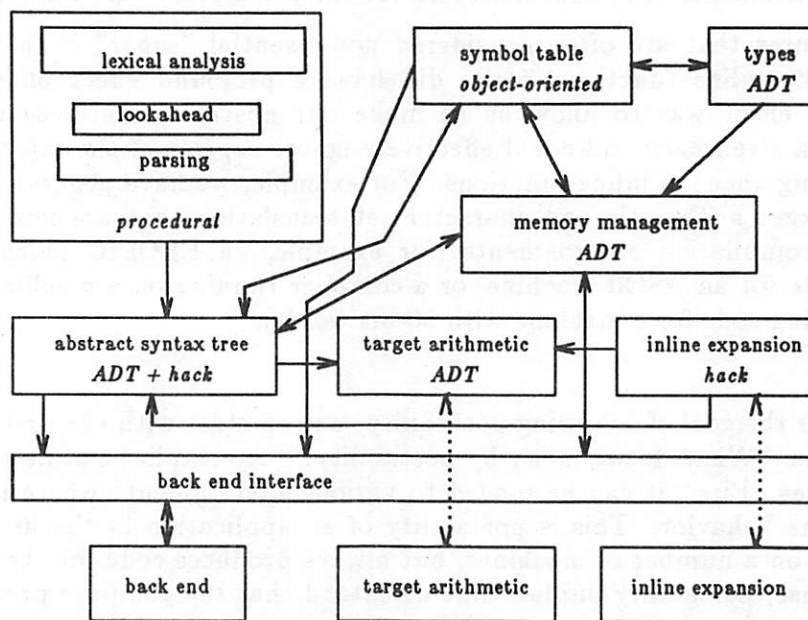


Figure 1: Compiler Structure

The general structure of the compiler is shown in Figure 1. The paradigm central to the design of each front end module is indicated. For example, the internal representation of type is implemented as an abstract data type, in that the implementation of the type is private and accessible only through a public interface. This proved to be an important property, as its implementation underwent several major redesigns in the course of development.

To cope with the complex nature of scope in C++, the symbol table is implemented as a dynamic graph of scope objects.<sup>[4]</sup> In this way, global, class, and function scope can be treated in a uniform way throughout most of the compiler, and block structure can be ignored for the most part as a detail of function scope. More importantly, the abstraction of the problem of the complex interactions of various kinds of scope to general interactions of generic scope objects led to a simple solution to problems of scope that extended easily to encompass not only multiple inheritance, but also compile-time memory management and approaches to avoid recompilation.

The design of the abstract syntax tree (AST) shows how C++ supports flexibility in melding paradigms. In general, a node in the AST is created by a constructor that takes as arguments the nodes that are the roots of its subtrees and produces a fully-elaborated subtree. However, in certain exceptional cases there is not enough information in a subtree to complete the semantic analysis.<sup>1</sup> Rather than provide a general (and potentially expensive) mechanism for dealing with these exceptional cases, we simply mark the nodes that are "unfinished" and fix them up later when more

1. For example, in the expression `p = &f`, where `p` is of type pointer to function and `f` is an overloaded function name, the instance referred to by `f` and therefore the type of the subtree `&f` can not be determined until reduction of the assignment expression.

information is available, in blatant disregard for the principles of data hiding.

Other features that are often considered non-essential "sugar" from a theoretical perspective--like inline functions--really did have a profound effect on the compiler design. Their effect was to allow us to make our abstract interfaces very general, knowing that a given back end could effectively ignore aspects of the interface it didn't need by defining vacuous inline functions. For example, we have general interfaces for performing target arithmetic and character set translation that are non-null only for certain cross-compilation environments (for example, an EBCDIC machine compiler generating code for an ASCII machine, or a compiler running on a machine with 32-bit words generating code for a machine with 60-bit words).

## Portability

To consider the goal of achieving portability, we can start with the first in a series of naive questions, "What do we mean by portability?" A compiler can be portable in at least two senses. First, it can be moved to various environments where it will exhibit similar runtime behavior. This is portability of an application in the usual sense; the compiler runs on a number of machines, but always produces code for the same target. In another sense, portability implies that we intend that the compiler produce code for different targets. In this case, by portability we mean retargetability. In the first sense, portability is not difficult to accomplish (at least approximately), so our approach was to reduce portability of the second sense to the first.

Most practical approaches to retargetable compiler design result in a front end that is tightly coupled to a retargetable code generator. This is, for example, the architecture used by the various pcc compilers for C.<sup>[5]</sup> Unfortunately, this generally (although not essentially) causes the compiler internals to converse in the language of the intermediate representation of the code generator, which is more suited to machine issues than to language issues. This complicates the task of recovering or producing source-level information for programming environment tools. Additionally, this arrangement requires that the designer of the code generator be clever enough to produce an architecture that is easily retargetable to a large class of machines while still generating good code with reasonable efficiency.

We chose our targets to be code generators rather than machines. In essence, by moving the target of the compiler from the hardware to a user program, the problems of retargetability become those of portability. Portability in this sense is a well-understood problem of software *engineering*, and as designers we do not have to resort to uncommon ingenuity to produce a portable compiler. This also allows us to utilize an internal representation that is tightly coupled to the source language, and it permits the design of a minimal interface between front and back ends.

One advantage of this approach is that it allows the use of existing code generators, no matter what their technology. Some of the best code generators (in the sense of speed of execution and quality of generated code) are hand coded, and they would be difficult to reimplement within the constraints of a more generally retargetable architecture. In addition, production code generators often provide more functionality than straightforward translation, and may also, for example, produce information for object file utilities or special code sequences to circumvent hardware bugs. In using existing code generators, the porter of the compiler is relieved from rediscovering and reimplementing these aspects of code generation. Because of the tight coupling of intermediate representation to source language, it is convenient under this scheme to

reflect front end information to generate code for special execution environments. For example, some application areas may require that all member functions of a given class be implemented as remote procedure calls, that data hiding be implemented by segmenting class members in distinct physical memory areas according to their protection, that heap allocation be accomplished through handles, or that certain data be persistent. In that these special semantics are logically part of the execution environment, they can be implemented simply as part of a code generator interface without resorting to modification of the C++ language.

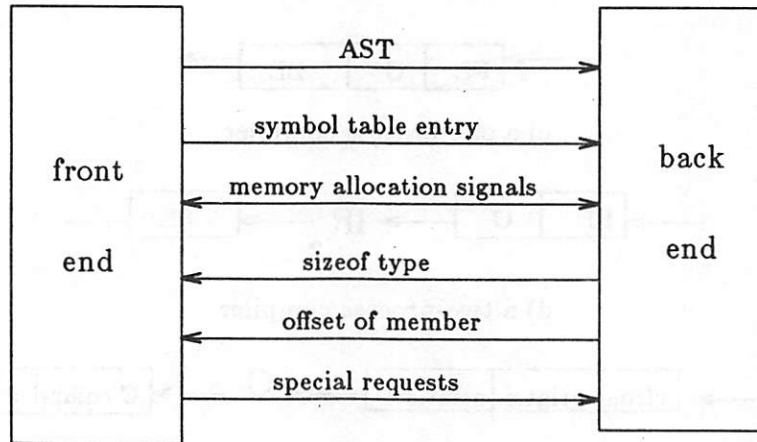


Figure 2: Back End Interface

The compiler considers the back end to be an abstract data type. The abstract interface, illustrated in Figure 2, provides two-way communication between front and back ends. The task of retargeting the compiler is that of implementing the abstract interface for an existing back end. This is no different from the task of portability in the traditional sense, and the problems of retargeting are reduced to those of porting. As mentioned above, because the source language is fixed the abstract interface, and therefore the effort required to port the compiler, is very restrained. This approach encourages porters of the compiler to "lay off" the front end, minimizing the unintentional introduction of changes to the language accepted by the implementation.

To put this approach to retargetability in perspective, consider the portable structure of cfront. Although cfront is usually described as using C as its assembly language, it can also be viewed as a 2-process compiler that uses C as an intermediate representation. The abstract structure of the compiler is similar, where the "glue" routines that implement the abstract interface perform the translation from the compiler's intermediate representation to the intermediate representation understood by the back end. Note that the interface makes no assumptions about the form of the intermediate representation understood by the back end. There are also few assumptions about the form of the back end itself; it can be part of the same process as the front end, a separate process, or a coroutine.

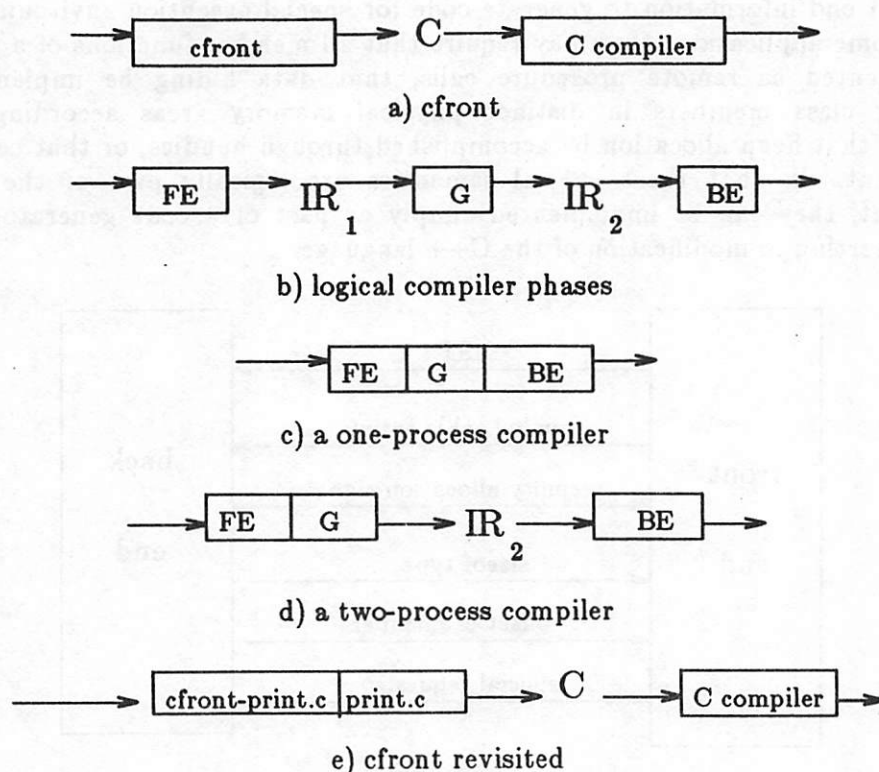


Figure 3: Compiler Configurations

The compiler typically functions as a single process, but can be trivially made two-process if hardware or software environment dictate. Looking more closely at cfront, we realize we have come full-circle. This must certainly imply something about the compiler's design, but I'm not yet certain what. Finally we note that compiling C++ includes preprocessing and assembly. Current and ongoing work will allow for single process compilation from unpreprocessed C++ to object by merging these functions into the compiler.

The compiler currently has back end interfaces for C (à la cfront), CG (pcc-based), and Nail (a functional machine description language). The effort required to write an interface from scratch for a new code generator is typically 1-2 staff weeks. For interfaces that can be constructed by modification of an existing interface the porting time is correspondingly less. Two of the current back ends, CG and Nail, are themselves retargetable and are shared with other portable compilers. Whenever one of these other compilers is ported to a new environment, the C++ compiler is automatically ported as well.

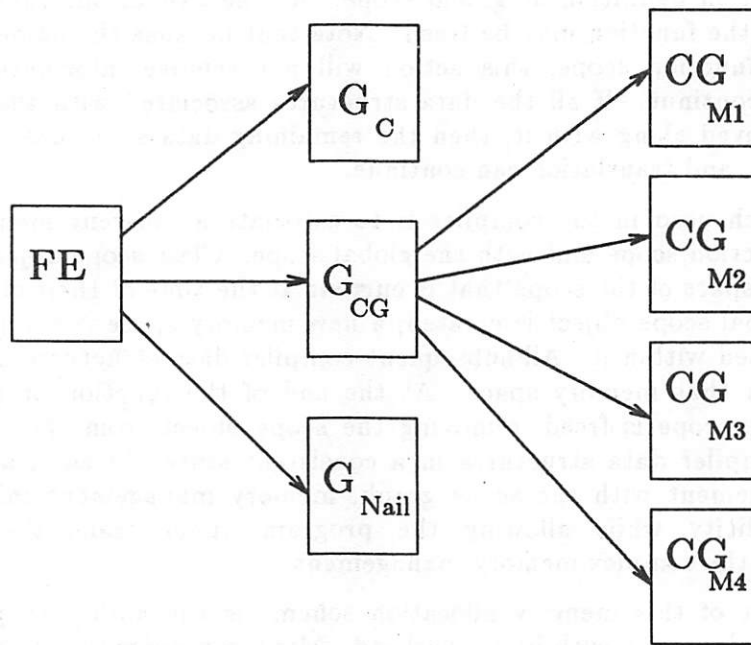


Figure 4: Leveraging the Compiler Front End

### Models of Compilation

The utility of a compiler is not determined only by the machines on which it runs and the machines for which it can generate code, but also by its flexibility to serve within a variety of models of compilation. In the previous section, we have seen how the compiler can be configured to perform source to object translations in one, two or more processes, but these do not really represent different models of compilation, but merely variations of a single model.

To understand better what we are trying to accomplish, we ask our second naive question, "What is compilation?" For a compiler writer the obvious<sup>2</sup> answer is, "Compilation is the process of moving the compiler from one internal state to another, with side effects (like code generation)."

Consider the implementation of the symbol table mentioned earlier in which scopes were encapsulated as objects.<sup>3</sup> The logical locality implied by an object also permits an easy implementation of physical locality, either as contiguous memory or in easily accessible locations. We also note that, if every language feature is associated with a given scope, then the compiler data structure used to translate an instance of that feature can be associated with a given scope object. The lifetime of the data structure should be identical to that of the scope object, just as the lifetime of the feature instance is no greater than that of the scope in which it appears. As an example

2. The answer became obvious about halfway through development of the compiler.

3. See [4] for more detailed discussion of the design of the compiler symbol table, and [6] for the implications of this design on the process of compilation.

consider a function definition at global scope. At the exit of the function scope, the scope object of the function may be freed. Note that because the name of the function is not in the function scope, this action will not remove information required for translation to continue. If all the data structures associated with the function scope object are removed along with it, then the remaining data structures will be left in a consistent state, and translation can continue.

The approach used in the compiler is to associate a different memory space with each active function scope and with the global scope. Class scope objects are allocated in the memory space of the scope that is current at the time of their creation. When a function or global scope object is created, a new memory space is set up and the scope object is allocated within it. All subsequent compiler data structures *within that scope* are allocated in that memory space. At the end of the function or global scope the memory for the scope is freed, removing the scope object from the scope graph and leaving the compiler data structures in a consistent state. In associating patterns of memory management with the scope graph, memory management inherits the scope scheme's simplicity while allowing the program under translation to drive the construction of the complex memory management.

A side effect of this memory allocation scheme is the ability to process multiple program files in sequence with little overhead. Many compilers (including early versions of the C++ compiler) process only a single file per invocation because, due to the way memory is managed, cleanup between files would be prohibitively expensive. The current version of the C++ compiler solves this problem by creating an object that represents the scope that exists before and between file compilations. Its memory space contains invariant data that is created on compiler initialization (but no names). At the end of each file the global scope object is deleted to re-initialize the compiler. In avoiding loading and initialization for each file, the compiler realizes significant gains in translation time for compilations that comprise several files. Once the preprocessing and assembly functions are merged into the compiler, this technique will allow a build comprising any number of files to be accomplished in two processes: one to compile and one to link edit.

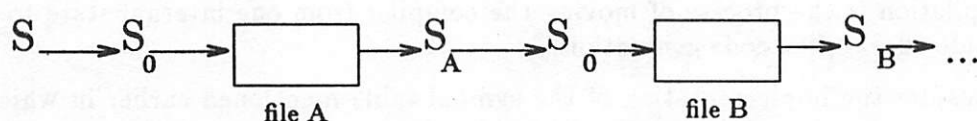


Figure 5: Reinitializing the Compiler

Other, as yet unimplemented, applications that could make use of this strong association between memory space and scope are perhaps best viewed as compiler support for programming environment tools. For example, this organization would simplify an implementation that minimizes recompilation of common file prefixes.

Consider the problem of compiling a sequence of files, each of which has a common initial part. If no external effect is produced during the compilation of the common prefix (such as code generation) then the effect of the translation to that point is to move the compiler from its initial configuration to a given compilation state. If this state can be restored easily, the common part need be compiled only once for the sequence of files.

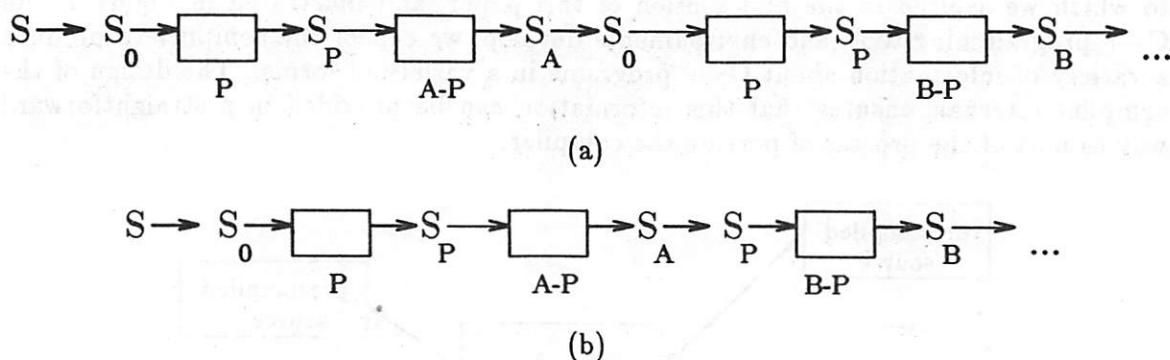


Figure 6: Avoiding Common Prefix Recompilation

In most compilers the state of compilation is represented entirely within the compiler's data structures. If the memory allocation scheme described above is used, these are contained within the memory spaces of a collection of scope objects. Thus, just as earlier we were able to restore the initial compiler state by deleting all scope objects but the initial "between file" object, it is possible to restore any intermediate compilation state through judicious segmentation and deletion of scope objects. (Note that there needn't be a one-to-one relationship between scopes and scope objects.) The ease with which this can be accomplished is highly language dependent. For example, the C++ compiler could easily implement delayed code generation (expanding the class of code prefixes it could handle) because much of that capability is required for inline function expansion. On the other hand, in C++ the definition of a name may be begun at one point in the compilation and added to at a remote point. (For example, a function prototype may add default argument initializers.) In order to handle prefixes containing definitions of this kind it would be necessary to record and undo effects of this sort that occur after the prefix.

Once the case of the single common prefix can be handled, a number of additional opportunities arise. Perhaps the most obvious of these is the stacking of compilation states. In this way the prefix dependencies of a set of files to be compiled can be arranged into a DAG and the files compiled in such an order that by pushing and popping the scope objects representing compilation states minimal recompilation is performed. Additional gains can be obtained by the ability to save an external representation of a compilation state in a program database. Although these compilation states could represent any piece of a compilation, they would generally represent compiled versions of header files or class definitions.

At this point we pose our final naive question, "What is a compiler?" Based on the above discussion, a good working definition would be, "A compiler is a program that changes its internal state in response to a representation of a program piece and produces a representation of some aspect of some portion of that program piece." This definition is general enough to cover a variety of translation situations, including traditional source-to-object translation. In addition, we have shown that the compiler might also read and produce precompiled objects. However, there are more useful views of a program than simply the (object code) representation of its execution semantics or the internal state it produces when processed by a compiler. For example, a source browser might require symbol table information so as to be able to map the use of an overloaded function identifier to its declaration. This is the general view of compilation

to which we aspired in the first section of this paper, and illustrated in Figure 7. As C++ programming tools and environments develop, we expect the compiler to produce a variety of information about C++ programs in a variety of forms. The design of the compiler internals ensures that this information can be provided in a straightforward way as part of the process of porting the compiler.

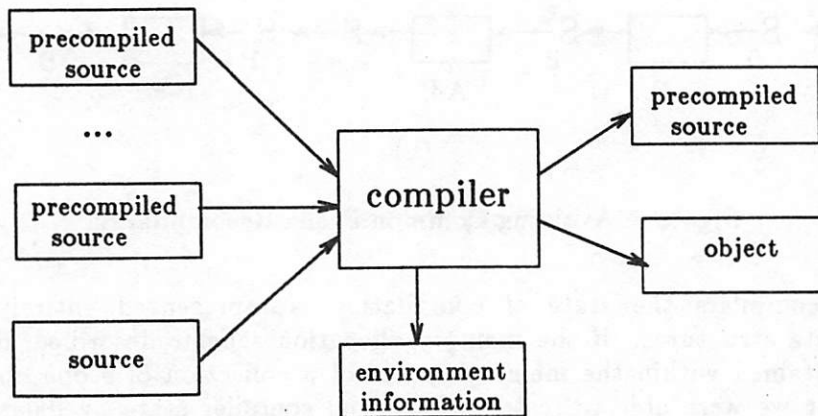


Figure 7: A Compiler

### Summary of Design Experiences

In many ways it seems silly to expound at such length on issues that are so obvious in retrospect, but if anything the "coming of age" in recent years of the data abstraction and object-oriented programming paradigms has shown us the power of thinking simply and directly about problems in software design. That is the theme of this paper.

There is nothing especially profound in noting that code generators should have simpler interfaces than machines and that most machines already have some kind of code generator running on them. It only requires a certain amount of laziness to avoid the whole problem of retargetability by reducing it to a problem in porting (and somebody else's problem, at that).

Naive questions about central issues and concepts also seem to be helpful. In many cases our terminology has failed to keep pace with our technology; as a result, the way we think about aspects of building and translating software is constrained. In this and other papers<sup>[4][6]</sup> we have examined the terms "scope", "symbol table", "compilation", and "compiler" from nontraditional viewpoints appropriate to the task at hand. As a result, we gained clearer insight into what we were trying to accomplish, and were able to extract a simple solution.

### Acknowledgements

Many thanks are due Sarah Hewins, Stan Lippman, and Kathy Stark for their careful reading and unbridled criticism of earlier drafts of this paper.

## REFERENCES

1. S. C. Johnson, course notes from SIGPLAN '82 Tutorial on Compiler Construction, June 21-22, 1982.
2. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.
3. *Draft Proposed American National Standard for Information Systems--Programming Language C*, X3J11/86-017.
4. S. C. Dewhurst, "Flexible Symbol Table Structures for Compiling C++", *Software - Practice & Experience*, Vol. 17 No. 8, August 1987.
5. D. M. Kristol, "Four Generations of Portable C Compiler", *USENIX Conference Proceedings*, Summer 1986.
6. S. C. Dewhurst, "Object Representation of Scope During Translation", *Proc. 1st European Conference on Object-Oriented Programming*, Paris, 1987.

# REFERENCES

1. S. C. Johnson, "A new algorithm for the fast Fourier transform," *IEEE Transactions on Audio and Electroacoustics*, vol. 28, pp. 17-26, 1980.
2. J. B. Evans, "The fast Fourier transform," *IEEE Transactions on Audio and Electroacoustics*, vol. 28, pp. 1-16, 1980.
3. J. B. Evans, "The fast Fourier transform," *IEEE Transactions on Audio and Electroacoustics*, vol. 28, pp. 1-16, 1980.
4. S. C. Johnson, "A new algorithm for the fast Fourier transform," *IEEE Transactions on Audio and Electroacoustics*, vol. 28, pp. 17-26, 1980.
5. J. B. Evans, "The fast Fourier transform," *IEEE Transactions on Audio and Electroacoustics*, vol. 28, pp. 1-16, 1980.
6. J. B. Evans, "The fast Fourier transform," *IEEE Transactions on Audio and Electroacoustics*, vol. 28, pp. 1-16, 1980.
7. J. B. Evans, "The fast Fourier transform," *IEEE Transactions on Audio and Electroacoustics*, vol. 28, pp. 1-16, 1980.
8. J. B. Evans, "The fast Fourier transform," *IEEE Transactions on Audio and Electroacoustics*, vol. 28, pp. 1-16, 1980.
9. J. B. Evans, "The fast Fourier transform," *IEEE Transactions on Audio and Electroacoustics*, vol. 28, pp. 1-16, 1980.
10. J. B. Evans, "The fast Fourier transform," *IEEE Transactions on Audio and Electroacoustics*, vol. 28, pp. 1-16, 1980.

# C++ for OS/2

John Carolan  
Glockenspiel, Ltd.

©1987 John Carolan.

## Section 1:

Why C++ is good for OS/2 and  
why OS/2 is good for C++

## Section 2:

If C++ is so wonderful  
why isn't everyone using it ?

## Section 3:

If there are problems with C++  
what's everyone doing to fix them ?

## Section 4:

Let's hear it for C++

## Appendix A:

C++ Sets - technology to the rescue

## Appendix B:

Progressive de-abstraction

UNIX is a registered trademark of AT&T.

IBM PS/2 is a trademark of IBM.

Microsoft, MS-DOS and OS/2 are registered trademarks of Microsoft.

CodeView is a trademark of Microsoft.

VMS is a trademark of DEC.

X Windows System is a trademark of MIT.

designer C++ is a trademark of Glockenspiel.

and so on

## Section 1:

**Why C++ is good for OS/2 and  
why OS/2 is good for C++**

The commercial market requires systems with

- comprehensive functionality
- distributed applications
- central databases
- high bandwidth human interfaces

IBM and Microsoft propose the solution

- use PS/2 hardware
- develop in C on OS/2

Indeed, OS/2 forms an excellent platform on which to construct the new applications.

Microsoft C 5.0 benefits developers because it is

- easy to assimilate
- easy to debug
- hardware specific
- highly optimized

C++ simply adds to these benefits.

Consider how C++ addresses the market requirements:-

## 1.1 Comprehensive functionality.

When software products get comprehensive they get more and more difficult to engineer. As you add features you add modules and you add dependencies between modules. C++ reduces dependencies by

- data abstraction
- code re-use
- strong type-checking
- encapsulation

and, more subtly, by encouraging better software design.

## 1.2 Distributed applications.

Distributed systems communicate most easily when the logical unit of communication consists of an object which exhibits the same functionality in each process or location. The C++ technique of encapsulating functionality with data representation makes transmissible functionality a natural part of the application architecture.

## 1.3 Central databases.

Data abstraction makes objects retrieved from a mainframe behave the same as objects occurring in the local systems.

## 1.4 High-bandwidth human interfaces.

Human interface selects very strongly in favor of C++ :-

- Objects in the interface map to objects in the software
- Sequences of member functions make up message processors
- The *Windows Virtual Machine* maps to a small number of system objects, making OS/2 easier to assimilate
- Data abstraction in the WVM makes C++ applications more portable to Windows 2.0 on MS-DOS and to X windows on UNIX.
- in-line functions improve the bandwidth

The Intel 286 architecture running in protected mode selects very strongly for C++ :-

Far objects consist of a code segment and a data segment which co-operate with the 286 inbuilt memory management. Once an object begins to execute, on-chip segment registers address both code and data, resulting in very fast access to RAM. Near objects cluster together to improve the performance of the address translation system. They also require shorter opcodes.

OS/2 supports very accurately IBM's PS/2 in its role as networked presentation server and C++ supports very accurately the construction of a set of applications to fulfill the PS/2's role.

Logically, C++ should become the most popular language for OS/2 development. However, some short-term problems must be overcome....

## Section 2:

**If C++ is so wonderful  
why isn't everyone using it ?**

Some short-term issues inhibit the growth of C++ on OS/2 :-

### 2.1 Poor marketing.

No-one broadcasts the clear benefits of C++.

C became successful in spite of relatively poor marketing, but UNIX still hasn't achieved the success it deserves, after ten years. The benefits of UNIX have never been correctly perceived nor correctly promoted.

### 2.2 Cost of assimilation.

Only one full book has yet been published on C++. Bjarne's book spends a lot of time discussing operator overloading, conversion functions, ways to defeat the heap manager, a gruesome implementation of generics using the C pre-processor and stream i/o. Although these features of C++ render helpful solutions in other kinds of software project, they are far from central in developing OS/2 applications. Bjarne's book spends little time on program architecture, program building and object-oriented design. This is not a criticism - Bjarne never intended that sort of book. However, the fact the book avoids those issues leaves the language still difficult to assimilate.

### 2.3 Bugs.

AT&T's C++ comes with \$2000 worth of bugs per copy. No verification suite exists for C++, so no way can be found for eliminating bugs quickly. C++ evolves as we speak, so new bugs will occur when we've fixed the present ones. People developing for OS/2 compare C++ to the Microsoft C compiler which contains very few bugs. They don't want an incorrect C++ translator, especially during the period while they struggle to become proficient in C++.

## **2.4 Speed of execution.**

Member function invocations take the address of the object and then execute by continuously dereferencing the pointer to the object. Under some circumstances, this is an overhead. If people use operator overloading and conversion functions, as per Bjarne's book, temporary variables creep into the code. The AT&T translator does not implement near or far objects, since it was never designed for Intel systems.

## **2.5 Debugger support.**

The AT&T translator provides no debugger support.

The previous problems apply mainly to C++, the next two apply also to C.

## **2.6 Portability.**

Even in C++, a product which exploits the OS/2 environment cannot later be ported to any other environment. People demand some technology for portability.

## **2.7 Product build.**

The common UNIX tools, such as make and sccs rapidly become inadequate for building huge systems. There comes a time when re-building a product is no longer an option. Instead, individual objects must be upgraded while the system is live. Currently, UNIX possesses no simple technology for in-flight object replacement.

## Section 3:

**If there are problems with C++  
what's everyone doing to fix them ?**

Most of the problems just described can be solved fairly simply.

### 3.1 Better marketing.

We can proclaim the benefits of C++

- by highlighting the C++ features for
  - data abstraction
  - object-oriented programming
  - strong type-checking
- by focusing on concepts rather than syntax
- by relating the concepts to OS/2 based systems
- by avoiding less relevant issues

Everyone at this workshop knows C++ to be a more enjoyable better designed language than C. If we wish to go on enjoying it we must portray its benefits clearly and concisely.

### 3.2 Quicker assimilation.

We need more books on C++, on object-oriented design. But most of all we need books which relate C++ to windows technology. The UNIX habit of dumping a language in the middle of the floor and letting people get on with it doesn't work in a business environment.

Ideally, people should be enabled to assimilate OS/2 concepts and C++ concepts in one smooth process.

We need courses, we need meetings, we need case studies.

All of these things can be achieved in 1988 !

**Glockenspiel** launched two training courses in September and we have a very urgent documentation program under way.

The rest of this section describes specific engineering projects also under way at Glockenspiel aiming to accelerate the acceptance of C++.

### 3.3 Correct C++.

To the problem of translator bugs, there is a quick fix and a slow fix.

The *quick fix* consists of a massive beta release program which has been underway since mid-summer. 30 sites, developing for Windows and OS/2 have beaten on the *designer C++* programming system for six months. Bug reports are down to less than one per month.

The beta release program costs a lot of money. It requires very meticulous bug tracking. Only when a particular beta release proves to be stable do we implement the bug fixes in the authentic release.

The *slow fix* consists of a formal specification for C++ followed by a verification suite followed by a re-write of the translator using good engineering practices. The slow fix will take more than a year to complete, but will leave the language in a very robust condition.

### 3.4 Code optimization.

The objective of the *designer C++* implementation on OS/2 is to out-perform hand-written C code. C++ maps elegantly onto the 286 hardware, so the way to get performance is to emit C code that coerces the C compiler to emit efficient object code. Very careful casting and the exploitation of the C compiler's hardware oriented features achieve our objective, as follows:-

#### 3.4.1 Pascal stacking sequence.

When functions have a fixed number of arguments, the pascal stacking sequence proves more efficient than the C stacking sequence. *designer C++* instructs the C compiler to emit pascal sequences for member functions with fixed length argument lists.

### 3.4.2 Object handles.

*designer C++* detects cases when the data representation of an object is precisely a 16-bit handle and passes the handle by value in member functions' first arguments.

### 3.4.3 Far and near objects.

For objects declared as near, member functions access their data using a 16-bit offset within the local segment.

For objects declared as far, member functions access their data using offsets embedded in the object code, relative to the data segment register.

### 3.4.4 Far and near functions.

Member functions may be declared far or near to improve the efficiency of the code emitted. Pointers to both functions and data may point at far or near addresses.

### 3.4.5 Far and near virtual functions.

Virtual functions may also be far or near. This is done by having more than one virtual function table per object. The virtual function table for far functions can be held in the code segment containing those functions, so only the 16-bit segment number need be stored in the object. The virtual function table for near functions can be held in the local code segment, so again, only a 16-bit offset is required. Further, the pointers within the virtual function tables themselves can all be 16-bit.

The current beta release of *designer C++* already performs over half of these optimizations.

### 3.5 The portable debugger.

*designer C++* optionally emits debugging code which calls a run-time debug package. On OS/2, the debug package becomes a shared library. Being a library protects it from trampling by the code being debugged. The interface to the debugger is public, so anyone can implement their own debugger or profiler. The same interface occurs on UNIX and VMS implementations of *designer C++*, so debuggers can be ported easily between environments.

Production versions of *designer C++* include the portable debugger code, but it is de-activated while beta testing of the system continues.

*designer C++* partially supports Microsoft's *CodeView* debugger. *CodeView* performs an excellent job on low-level problems but it was never intended to be object oriented.

### 3.6 Portability.

The strategy that makes applications portable between OS/2, Windows 2.0 and X windows relies on isolating non-portable code within a small number of modules. Program architecture differs from the usual UNIX application as follows:-

#### 3.6.1 The Windows Virtual Machine.

We re-phrase the OS/2 system call interface in terms of C++ objects, such as windows, fonts, brushes, files, messages, tasks etc. Just as one might implement a UNIX file system on VMS by providing UNIX system calls in the VMS run-time library, we implement the *Windows Virtual Machine* on each different operating system. Code which uses the WVM interface ports fairly easily from one environment to another.

#### 3.6.2 Objects.

The user designed objects should contain no code which does not port trivially from one system to another. Typically, users design objects like documents, paragraphs, spread-sheet cells, records, indexes, keys, packets etc. and they derive objects from these and from the ones in the WVM.

### 3.6.3 Sets.

*Sets* form the third element in the portability scheme. In simple applications, the programmer chooses to employ named objects, which get constructed on the stack or in static data segments. Non-trivial applications require arbitrary collections of anonymous objects. Sets are just containers for anonymous objects. An object qualifies for membership of a set if it belongs to the class hierarchy over which the set is declared.

Appendix A to this paper concerns the functionality of sets.

Sets are just a convenient hat stand where these functions can choose to hang their hats. The designer of any particular set decides which kinds of functionality to provide. Sets should incur no overhead caused by catering for functions they don't need.

The *designer C++* system declares sets as classes, it does not alter the C++ language in any way.

Sets abstract system dependencies.

Both the code which makes use of the set and the objects the set contains ought to be portable. The functionality of sets consists of functions that apply to collections of similar objects.

Briefly, the following functions fall into this category:-

- heap management
- object transmission in a distributed system
- concurrent execution of similar objects
- message processing
- managing persistent objects
- virtual memory optimization
- object version control in an upgradeable system

For example,

Heap management differs totally between UNIX, MS Windows and OS/2, so the inherently non-portable set assumes responsibility for heap management rather than the inherently portable object hierarchy governed by the set.

Objects which managed their own memory by poking their 'this' pointer could not be portable between systems. Far less could they be candidates for re-usable code !

Of all the short-term solutions to the problems facing OS/2 developers, sets requires the most careful design. Most of the functionality has been prototyped, but the final version will not ship until Q2 '88.

### **3.7 Declaration database.**

The declaration database alleviates the problems encountered while building large, complex systems on UNIX. The project must group header files into groups with serial dependency. When someone touches a header file the make system only processes the group to which it belongs. The *designer C++* system processes header files onto the declaration database. Each C++ file states which header file groups it depends on. The *designer C++* system processes only files dependent on the touched group. *designer C++* only emits C code for those type declarations actually invoked in the module's C++ code.

With debugging on, each object includes its type-name and the debugger accesses the database to interpret the object correctly.

Unlike sets, the declaration database was simple to design. It should also ship Q2 '88.

**Summary: All of the problems which inhibit the acceptance of C++ by OS/2 developers can and will be fixed by mid '88.**

## Section 4:

### Let's hear it for C++

Not all the problems with C++ are technical, nor are all the solutions technical.

On the technical front,  
AT&T's C++ needed some improvement before OS/2 developers invariably chose it over C.

*designer* C++ already goes a long way towards meeting the market requirements.

*Sets* do not provide an off-the-shelf solution to engineering problems, but they do provide a blue-print by which effective, consistent, portable, high-performance applications can be built for new environments.

C++ offers such clear benefits for people developing large systems and for people developing human interface software that it deeply deserves to be the most popular language for OS/2.

What C++ needs most is energetic well-informed promotion.

## Let's hear it for C++ !

## Appendix A:

### C++ Sets - technology to the rescue

#### A.1 General functionality.

The type declaration for a set usually begins:-

```
class SetofFoo : public Set  
{  
.....
```

*Foo* is the base of a derived class hierarchy.  
So,

```
SetofFoo myset;
```

declares *myset* to be a set of *Foo*-type objects.

##### A.1.1 Serial sets.

A *serial set* appears to have these member functions:-

```
void New( Typename ); // actually a macro  
void Delete();  
bool Begin();  
bool End();  
bool Next();  
bool Prev();  
bool Nth( unsigned );  
void Exec( void ( Foo::* )() );  
void Every( void ( Foo::* )() );  
unsigned Count();  
unsigned Rank();
```

<i>New</i>	constructs a new object.
<i>Delete</i>	destroys the current object.
<i>Begin</i>	selects the first object as current.
<i>End</i>	selects the last object as current.
<i>Next</i>	increments the current object number.
<i>Prev</i>	decrements the current object number.
<i>Nth</i>	selects the <i>nth</i> object, counting from zero.
<i>Exec</i>	executes the <i>Foo</i> member function for this object.
<i>Every</i>	executes the <i>Foo</i> member function for every object.
<i>Count</i>	returns the number of live objects.
<i>Rank</i>	returns the number of the current object.

Typical uses of *myset* might be

```
myset.Begin();
while( myset.Next() )
    Exec( Foo::print );
```

If the order of printing did not matter,

```
Every( Foo::print );
```

would do the same job.

### A.1.2 Ordered sets.

*Ordered sets* provide an ordering function:

```
int Order( Fooiden );
```

where *Fooiden* is some object, usually a string, which could identify a *Foo*.

The *Order* function returns the same values as *strcmp*.

Ordered sets also have a *Find* function:

```
bool Find( Fooiden );
```

Which finds the object identified by *Foo*iden. If *Find* is unsuccessful, following it immediately by a *Next* or *Prev* selects the closest match in either direction.

Even for ordered sets, *Every* does not guarantee any particular order of execution.

Clearly, sets are abstract. The implementation of a particular set could be an array or a linked list or a file.

Just as in the case of the portable debugger, the interface to sets is public property, so anyone can provide new implementations which just plug into a program.

Most sets will provide no more functions than the ones described above.

We have chosen this precise interface to preserve the extensibility of class hierarchies while supporting very fast execution.

## A.2 Heap management.

The member functions of *set* neither expect nor return pointers. Pointerless code offers less opportunities for error. However, the principal reason for not using pointers is to avoid dependency on particular heap management systems. Microsoft Windows, for example, has two heaps and can move objects on the heaps if it needs the space. The *Exec* function on a MS Windows system operates by locking the object in position, executing the member function and unlocking the object again. Movable heaps also cause *New* to be a macro. The set must allocate heap space for the object and lock it, then call the constructor, then unlock the object again. Pointerless code also circumvents the differences between UNIX and OS/2. OS/2 global heap manager returns segment numbers rather than pointers.

We considered solving the heap management problems by re-implementing the dot and arrow operators in C++. The proliferation of different heap mechanisms within the same environment - Microsoft Windows - forced us to abandon an operator overloading approach in favor of the more abstract set mechanism.

It is common to have several sets of the same type of object, each set with its own heap management. Functions such as *New*, *Delete* and *Exec* cause the C++ heap manager to select the OS/2 heap manager appropriate to the particular set.

A further use of pointerless processing of a set is the implementation of shared objects. The locking mechanism works similarly to the movable object mechanism.

### A.3 Distributed objects.

Given a class declaration, the *designer C++* system will emit a template for representing the data part of the class on external media. This layout is system independent. It can be picked up by the set's data transmission functions.

### A.4 Parallel processes and concurrent execution.

The *Every* function does not guarantee any order in which the set will be processed. OS/2 supports light-weight processes, so it is possible to code the *Every* function so that it executes the *Foo* member function concurrently for several *Foo* objects. In a distributed application or a parallel processor, *Every* could execute the member function on different systems.

### A.5 Message processing.

Messages drive Windows and OS/2 applications. A continuous stream of messages repeatedly activates each Window process which in turn delegates the processing to the objects or sets of objects that it owns. Corresponding to certain messages, the set selects a procedure for dealing with each message, depending on the state of the application. These procedures normally consist of a sequence of member function executions for every object in the set. So each set has one or more message processing modules. Messages tend to be system dependent and tend to need action by every object in a set. That is why the set takes responsibility for message processing.

### A.6 Persistent objects.

Applications frequently need to save the state of a set of objects and restore it later. The set does this by 'transmitting' the objects to a central saver process, just as it would do for an application with distributed objects. Similarly, the later process 'receives' the stream of objects from a central restorer process.

## A.7 Virtual Memory support.

On systems without virtual memory, the pointerless operation of sets can be used to implement a virtual memory system. On systems such as OS/2 which do have virtual memory, sets make its operation more efficient. All the data required for the navigational aspect of the set - the operation of *Find* and *Begin*, for instance - can be held in one segment. Navigational operations on the set do not need the bodies of objects to be present in physical memory. Even when an object is invoked, its code is all in one segment and its data is all in one segment, minimizing the number of fetches. Objects on local heaps behave even better, in that the virtual address of all objects on the local heap occur within 64K of each other.

## A.8 Object version control.

Where several systems in a network run different generations of an application or where new modules are loaded dynamically on the same system, the responsibility of upgrading and downgrading objects between generations is again the responsibility of each set. The communications functions of the set check the generation number of each inbound object and pre-process it during the construction process.

## Appendix B:

### Progressive de-abstraction.

*Sets* classes are nothing but an abstraction of C style container objects such as lists, files or arrays. If the developer starts by expressing a collection of objects as a linked list, say, code developed as a client of the collection and the objects it collects tends to become very dependent on the specific linked list implementation. That sort of dependency is just what data abstraction strives to avoid. We find that the best methodology for C++ is *progressive de-abstraction* :

You start with the most abstract version you can devise of any object. Then you measure its impact on performance in a prototype. Based on feedback, you de-abstract the object until the performance comes within budget.

Put another way, it takes just a little work to de-abstract an existing object, because you don't break client code. But it takes a lot of work to make a not-very-abstract object more abstract.

Law of entropy:

**In any software system, data abstraction decreases.**



# C++ on the Macintosh

Ken Friedenbach, Ph.D.  
Development Systems Group  
Apple Computer, Inc.

## Abstract

This paper discusses a C++ implementation based on CFront for use in developing Macintosh applications. In addition to the compiler, header files and libraries provide access to Macintosh operating systems and toolbox functions. The paper also describes C++ header files for MacApp, the extensible Macintosh application written in Object Pascal. These header files allow the writing of MacApp applications in C++.

Some Apple extensions to the C++ language are described which allow full integration with the Macintosh run-time environment, the current MPW C compiler, Object Pascal, and MacApp.

C++ and Object Pascal are compared with respect to language features and current implementation optimization. Some issues related to C++ compatibility with ANSI C are discussed. Finally, a few words are devoted to directions for future development for C++ and object oriented languages.

## 1. Introduction

This paper discusses work in progress on a port of the AT&T CFront compiler [1] to the Macintosh Programmer's Workshop (MPW) development system. [2] This paper does not discuss support for C++ on A/UX, Apple's version of UNIX for the Macintosh II.

**1.1 Objectives.** There are several primary objectives Apple hopes to achieve in providing C++ for the Macintosh. First, C++ will provide a standard object oriented language for C programmers developing applications for the Macintosh. Second, Apple will provide C++ interfaces to all the Macintosh operating system and toolbox functions. Third, the implementation will allow mixing Object Pascal and C++ in application programs. Finally, building on the ability to mix C++ and Object Pascal, Apple will provide C++ interfaces to MacApp, the extensible Macintosh Application.

**1.2 Reasons for choosing C++.** C++ was chosen over other C-like object oriented languages because of a number of factors. First, it is highly compatible with both C and ANSI C, allowing a smooth evolution from existing sources and compilers toward better language definitions and newer compilers. Second, the CFront compiler from AT&T was relatively complete, had been tested in production use, and was designed for portability to any environment which supports C. And third, the member function and derived classes of C++ provided a close semantic match for interfacing with the methods, inheritance, and overriding of objects in Object Pascal.

Section 2 contains a description of the Macintosh Programming Workshop (MPW). [2] Section 3 contains some examples of Object Pascal code. [3, 4] And Section 4 contains an overview of MacApp, the extensible Macintosh application which is written in Object Pascal. [5] These sections are very brief. They only introduce a few concepts which may help the reader in understanding the relationships between C++, Object Pascal, MPW, and MacApp. For more information, consult the references.

Section 5 describes the Apple extensions to C++. [6] Section 6 compares C++ and Object Pascal. Section 7 mentions some directions for future work.

## 2. The Macintosh Programmer's Workshop (MPW).

The MPW development system is based on an integrated shell/editor environment with a standard Macintosh user interface. The editor provides for multi-window editing of text files, with the usual cut, copy, and paste commands.

The shell implements a UNIX-like command language. The command language has syntax for re-direction of input, output, and standard error files, including pipes and re-direction to windows or selections within windows. At both the command language and program level, it is transparent whether a text file is open in a window, or is closed and resides on disk.

The command language provides for shell variables, alias substitutions, symbolic references to parameters, return codes, and conditional and loop constructs. The shell has built-in commands for performing edit functions, filing operations, operations on windows, and operations on menus. All MPW commands can be executed in scripts or can be invoked interactively, with menu selections, or with function keys.

Standard development tools are run as separate tasks within the integrated shell/editor environment. MPW tools include compilers, an assembler, a linker, make, and dozens of other development tools.

MPW is an attempt to combine the best features of a UNIX-like development environment with the Macintosh user interface.

## 3. Object Pascal

Object Pascal consists of object oriented extensions to Pascal which are reminiscent of Smalltalk-80 or Simula-67. Object definitions are similar to Pascal records, but contain both data fields and procedures or functions (called methods). Objects are declared in tree structured inheritance hierarchies. A descendent object definition inherits the data fields and methods of the ancestor object definition. However, a descendent object definition is allowed to override the inherited method, provided that the parameter list is the same.

As a convenience in maintaining a large inheritance hierarchy, an overriding method is allowed to call the inherited method without knowing exactly which ancestor is the closest implementer. And as a security to avoid inadvertent overriding in a large inheritance hierarchy, the keyword `override` is required.

As an example of objects in Object Pascal, consider the inheritance hierarchy shown in Figure 1, which is taken from a small sample program in the MacApp manual. [5] TObject is the root type for TShape and all other MacApp objects. TObject provides a common set of storage management operations. Two levels of allocation and freeing are provided. This allows MacApp to control (and hide) most details of managing objects. And the user can still override, to control the semantics of copying objects which contain references to other objects.

## TYPE

```

TObject = OBJECT
  FUNCTION ShallowClone: TObject;    { Simply copies the object data. }
  FUNCTION Clone: TObject;
    { Defaults to calling ShallowClone. Override to copy objects referred to by fields. }
  PROCEDURE ShallowFree;              { Simply frees the object data. }
  PROCEDURE Free;
    { Defaults to calling ShallowFree. Override to free objects referred to by fields. }
  END; { TObject }

TShape = OBJECT (TObject)
  boundRect: Rect;                    { bounding box }
  PROCEDURE TShape.IShape;             { Initialize fields }
  PROCEDURE TShape.RandomRect;         { Assign a random rectangle to boundRect }
  PROCEDURE TShape.Move;               { Assign another random rectangle to boundRect }
  PROCEDURE TShape.Draw(pat: Pattern);
  PROCEDURE TShape.Erase;
  END; { TShape }

TArc = OBJECT (TShape)
  startAngle, arcAngle: INTEGER;
  PROCEDURE TArc.IArc;                 { Initialize fields }
  PROCEDURE TArc.Draw(pat: Pattern); OVERRIDE;
  PROCEDURE TArc.Erase; OVERRIDE;
  END;

TRoundRect = OBJECT (TShape)
  ovalWidth, ovalHeight: INTEGER;     { curvature of rounded corners }
  PROCEDURE TRoundRect.IRoundRect { Initialize fields }
  PROCEDURE TRoundRect.Draw(pat: Pattern); OVERRIDE;
  PROCEDURE TRoundRect.Erase; OVERRIDE;
  END;

```

Figure 1. Examples of object definitions in Object Pascal

The above definitions define a hierarchy of object types which are related as follows:

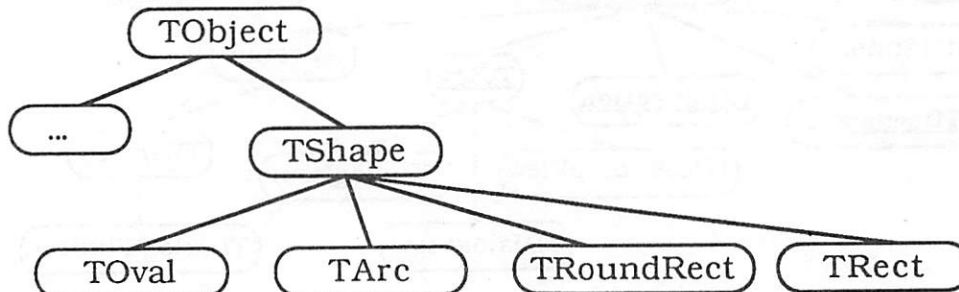


Figure 2. Inheritance Hierarchy of object definitions.

TShape is an abstraction from the specific shapes supported: TOval, TArc, TRect, and TRoundRect. In the original application TShape, TArc, TRect and TRoundRect were implemented in Object Pascal. In this paper, we describe how TOval is implemented in C++.

At the level of TShape, common operators are defined. For individual shapes, specific operators are implemented. Figure 3 shows the implementation of two operators: RandomRect, which is common to all objects of type TShape, and Draw, for objects of type TRoundRect.

```

PROCEDURE TShape.RandomRect;
  CONST width = 40; height = 40; minVerticalPos = 75;
  VAR rand1, rand2: INTEGER;
BEGIN
  rand1 := ABS(Random) MOD (screenBits.bounds.right - width);
  boundRect.left := rand1;
  rand2 := ABS(Random) MOD (screenBits.bounds.bottom - (height + minVerticalPos));
  boundRect.top := rand2 + minVerticalPos;
  boundRect.right := boundRect.left + width;
  boundRect.bottom := boundRect.top + height;
END; { TShape.RandomRect }

PROCEDURE TRoundRect.Draw (pat: Pattern);
BEGIN
  FillRoundRect(boundRect, ovalWidth, ovalHeight, pat);
  FrameRoundRect(boundRect, ovalWidth, ovalHeight);
END; { TRoundRect.Draw }

```

Figure 3. Examples of method implementations in Object Pascal.

Individual instances of an object definition share code for the methods and the dispatch tables for calling the methods at run-time.

#### 4. MacApp, an extensible Macintosh application

MacApp is a skeleton application which defines and implements the hierarchy of Object Pascal types shown in Figure 4. The object definitions shown in bold are the ones a user normally must customize to implement a MacApp application. The other object definitions may have one or more methods that a user might override to obtain customized behavior.

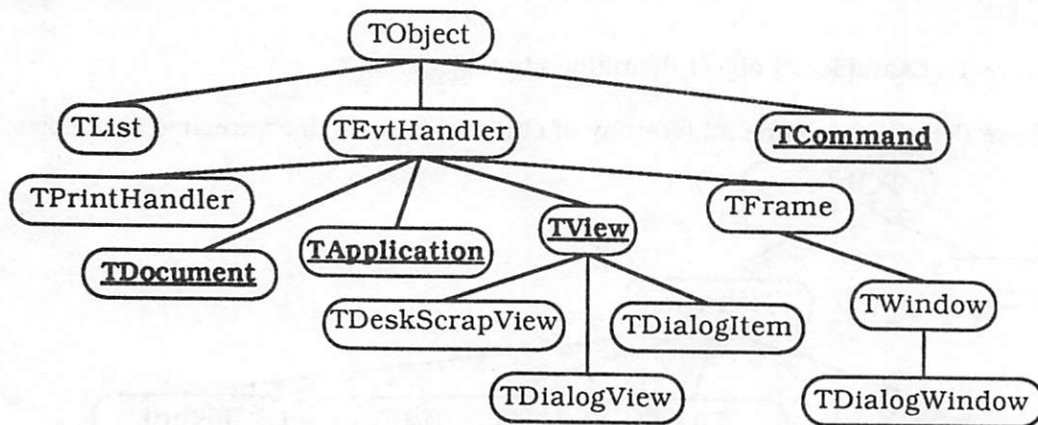


Figure 4. Part of the object hierarchy in MacApp.

Part of a simple minimum application which overrides the appropriate methods is shown in Figure 5.

```

TMiniAppl = OBJECT (TApplication)
  PROCEDURE TMiniAppl.IMiniAppl (itsMainFileType: OSType); { Initialize globals. }
  FUNCTION TMiniAppl.DoMakeDocument (itsCmdNumber: CmdNumber): TDocument;
    OVERRIDE; { Launch a TMiniDoc when application is started, or for New or Open. }
END;

```

```

TMiniDoc = OBJECT (TDocument)
    fMinView: TMiniView; { A reference to the view created by DoMakeWindows. }
    PROCEDURE TMiniDoc.DoMakeWindows; OVERRIDE;
        { Launches the window representing the document on the screen. }
    PROCEDURE TMiniDoc.DoMakeViews (forPrinting: BOOLEAN); OVERRIDE;
        { Launch the view seen in the documents window; also called for printing. }
    END;

TMiniView = OBJECT (TView)
    PROCEDURE TMiniView.Draw(area: Rect); OVERRIDE;
        { Draw the view seen in the window, or being printed. }
    END;

```

Figure 5. Definitions for a minimum MacApp application.

## 5. Apple extensions to C++

There are three areas where C++ has been extended for the Macintosh: numerics, Pascal calling conventions, and support for Object Pascal.

**5.1 Numerics.** The Standard Apple Numerics Environment (SANE) is an IEEE-754 compatible environment that Apple has supported on all of its machines: the Apple II family (including the Apple III) and the Macintosh family (including the Lisa). Support of SANE implies adding some new numeric types to C++:

- extended** (an 80-bit or 96-bit floating point type)
- comp** (a 64-bit integer type)

In turn, the **complex** class from C++ has encouraged the numerics group to add a complex data type to the SANE specification.

There are some compatibility issues with both C++ and ANSI C with regard to IEEE floating point. The fundamental problem is that there now exists good hardware which supports only extended precision operations. Both ANSI C and C++ still prefer double precision, and do not allow extended precision everywhere, e.g. in arguments and function return values in libraries. At present, Apple is using extended precision in libraries (including stream, in, and out) for both speed and accuracy.

**5.2 Pascal conventions.** For historical reasons, the Pascal run-time conventions on the Macintosh differ from the standard C conventions:

- Arguments are pushed in the opposite order.
- Function values are returned on the stack, not in registers.
- Arguments are removed by the called function, not the caller.
- Register saving conventions are slightly different.
- Strings are represented in different formats.
- Pascal passes small structures by value, e.g. point.

In order to support these differences, and still allow mixing of Pascal and C code, each language has been extended to recognize external definitions in the other language.

The present MPW C compiler allows ANSI prototypes, but without very strong type checking. Only the size of arguments is used, in accounting for stack space. In this C, a Pascal procedure might be declared as follows:

```
void pascal SomeProc (short x, window * wp);
```

Earlier versions of C on MPW supported a similar extension of old style C functions:

```
void pascal SomeProc (x, wp)
    short x;
    window *wp;
    extern;
```

The C compiler supports both sets of calling conventions. Similar syntax exists in Pascal for declaring external functions which follow the C calling conventions. The C compiler also supports compiling a C function to meet Pascal conventions, so operating system and toolbox functions, which have a specified Pascal interface, can be implemented in C.

Some support has been added to CFront for this use of `pascal` as a storage class, and for the use of `void` as a return type. In addition, Macintosh operating system and toolbox functions are called directly with A-Traps. The C compiler supports special semantics for the following syntax:

```
pascal void DrawText (Ptr textBuf, short firstByte, short byteCount)
    { asm (0xA885); }
```

Earlier versions of C on MPW supported a similar extension of old style C functions:

```
pascal void DrawText (textBuf, firstByte, byteCount)
    Ptr textBuf;
    short firstByte;
    short byteCount;
    extern 0xA885;
```

The semantics of this construct is as follows: First, push arguments on the stack, as if calling a Pascal procedure. Then, in place of normal subroutine call, emit the indicated constant (an A-Trap instruction).

This `asm/extern` A-Trap syntax can be viewed as access to machine level conventions, similar to the in-line assembly language construct used with other operating systems. It is not clear what will happen in the future to this extension. But in any event, the syntax occurs only in Apple specific header files, and is transparent in the client code, since the A-Trap handler simulates a normal subroutine call.

**5.3 Support for Object Pascal.** As indicated above, all MacApp objects are derived from TObject. A corresponding definition in C++ is:

```
struct TObject : indirect {
    virtual pascal TObject *ShallowClone ();          /* Simply copies the object data. */
    virtual pascal TObject *Clone ();
    /* Default calls ShallowClone. Override to copy objects referred to by fields. */
    virtual pascal void ShallowFree ();               /* Simply frees the object data. */
    virtual pascal void Free ();
    /* Default calls ShallowFree. Override to free objects referred to by fields. */
};
```

Figure 6. TObject defined in C++.

Notice the use of the new keyword `indirect`. Indirect might be viewed (especially for compatibility) as a predefined, empty base struct:

```
struct indirect {
};
```

But in the Macintosh implementation of objects, `indirect` has special semantics. It is used to indicate that the structure or class will be named, allocated, and accessed in the

same manner as objects in Object Pascal. Virtual member functions of indirect classes are called via the method dispatch techniques described in Section 6.2. In the present implementation, this means that the objects are allocated on the heap and accessed via a handle (a pointer to a pointer). But these details are filled in by the compiler, and hidden in the source code. Naming conventions for indirect classes match Object Pascal and are significant to the linker. These names trigger optimization which is described below.

The definition of TShape in C++ is shown in Figure 7.

```
struct TShape : TObject {
    Rect boundsRect;
    virtual pascal void IShape();           // Initialize fields
    virtual pascal void RandomRect();       // Assign a random rectangle to boundRect
    virtual pascal void Move();             // Assign another random rectangle to boundRect
    virtual pascal void Draw(Pattern);
    virtual pascal void Erase();
};
```

Figure 7. TShape defined in C++.

Since TShape is derived from TObject, it also is indirect. Finally, we show the definition and implementation of TOval in C++ in Figure 8.

```
struct TOval : TShape {
    virtual pascal void Draw(Pattern);
    virtual pascal void Erase();
};
pascal void TOval::Draw(Pattern pat)
{
    Rect tempRect;
    tempRect = boundsRect;
    FillOval(&tempRect, pat);
    FrameOval(&tempRect);
}
pascal void TOval::Erase()
{
    Rect tempRect;
    tempRect = boundsRect;
    EraseOval(&tempRect);
}
```

Figure 8. TOval defined and implemented in C++.

## 6. Comparison of C++ and Object Pascal

This section discusses C++ and Object Pascal from the point of view of language design. It also summarizes the arguments for and against supporting indirect classes.

**6.1 Classes in C++ are more orthogonal than objects in Pascal.** In C++ functions are allowed as members of structs or classes. Like normal structs, classes can be global variables, automatic variables, or elements of an array.

In Object Pascal, objects can only exist individually on the heap and are accessed through a handle. This is a severe limitation of Object Pascal, which limits the efficiency of objects and the usefulness for problems which require more than a hundred or so objects. In practice, it also makes the mixing of normal Pascal code and objects more difficult than in C++.

In Object Pascal methods are only allowed as fields in object definitions. Object Pascal, like Pascal, does not support first class pointers to procedures as types and values.

**6.2 Methods in Object Pascal are simpler and easier to use.** In C++, a function must be declared as either virtual or not virtual (normal member) in the source code. In Object Pascal all methods are essentially virtual. But the linker performs optimization which reduces methods with single implementations to being normal procedures. This is important for large systems of software, since it preserves the ability of individual developers to override all standard implementations without paying a uniform run-time cost.

Methods in Object Pascal are called by a reference which is a pair of names: (methodName, objectType). The methodName is static, and is bound at compile time. But the objectType can be dynamic, and is obtained at run-time. (Typically, it is SELF, which corresponds to this in C++, but it can be obtained from explicit references to ancestor types, or from the keyword inherited). Object Pascal makes use of dispatch table factoring. Only functions implemented at each level appear. Conceptually, the process of calling a method is performed as follows:

Beginning with the method-dispatch table for the objectType specified, look for the corresponding methodName. If the methodName is not found, then look in the method-dispatch table of the parent objectType. Continue up the chain of ancestors until the methodName is found.

This is essentially the method dispatch mechanism used in Smalltalk. It might seem to be fairly inefficient. However, the linker provided with MPW performs the following optimization:

At link time, perform a global analysis of objectTypes and methodNames. For any methodName which is only implemented once, substitute a normal subroutine call, and by-pass the method dispatch mechanism entirely. For methodNames which have more than one implementation, construct a new method dispatch table which contains a list of all the implementers.

With the above optimization, most method calls become regular subroutine calls. Where the run-time binding is used (as in the multiple versions of Draw in the examples), only one table needs to be searched.

In C++, without special compiler and linker support, the burden of choosing between normal member functions and virtual member functions has been placed on the programmer. And unfortunately, the burden has been placed on the wrong programmer, the designer of the base classes, not the client programmer. The client might decide to override a function the original designer did not expect to have overridden, and in a way that requires run-time binding. Or the client might decide to only use only a few of many virtual implementations provided.

The keyword virtual, like the keyword indirect, can be viewed as a pragma to a strong compiler which is compensating for a weak linker. Both may disappear when the back-end C compilers support ANSI C, and the standard system linkers become more intelligent about doing global analysis and optimizing run-time procedure tables.

### **6.3 Indirect is good: hiding the details of Macintosh memory management is good.**

Currently, objects in Object Pascal and indirect classes in C++ are the only language constructs which are high enough to effectively hide the details of memory management from code for data access. In using the normal declarations for accessing Macintosh operating system and toolbox structures, code becomes quite specific in dereferencing pointers once and handles twice.

The Macintosh memory management scheme has been a good one, in terms of packing a lot of functionality into limited memory sizes. However, it is a radically different scheme from traditional memory management, and it is likely to want to change when MMU's become widely available for personal computers with large memory sizes.

### **6.4 Indirect is bad: the cost of hiding the details of memory management is too high.**

Indirect classes in C++ should not be supported, because they are contrary to the intent of C. Hiding a level of indirection for something as simple as a data reference, can lull the C++ programmer into writing inefficient code. This is especially true for a processor such as the 68000 which has 32-bit addresses and only a 16-bit data bus. In following three levels of indirection (one for accessing the parameter *this*, and two for dereferencing a handle) the processor typically does at least nine memory references (three instructions and six data fetches).

In fact, the high cost of indirection was of great concern from the earliest days of Object Pascal (Classcal on the Lisa). Implicit references to *SELF* and the dereferencing of an object handle were carefully added to the register optimization of the Pascal compiler.

## **7. Directions for the future**

This section is partly a wish list and partly some indication about where Apple is heading in the area of object oriented languages.

Apple has defined a subset of C++, called "Minimal C++", [6] which it intends to use in implementing the interfaces to MacApp. Apple is encouraging developers of C compilers for the Macintosh to support minimal C++ as a standard way of providing object oriented features in current C and future ANSI C compatible compilers.

Apple would like to see improved compatibility between ANSI C and C++. But it is not always obvious which language is correct. Some areas of concern:

- The identifier name spaces could be made the same.

- Agreement about the status of tags in struct and union declarations.

- The const construct has a few problems, e.g. named expressions as array bounds.

- Function declarations with no arguments or a variable number of arguments differ.

C++ could benefit from a native code compiler and from linker support for optimization similar to those done by the MPW linker. While the run-time binding of function calls provides many benefits, it is not desirable to pay the cost of such a mechanism everywhere. Nor is it desirable to make the decisions in the source, since the decision cannot be made correctly for all users of large complex class hierarchies, such as the Macintosh and MacApp.

Inspired by C++, expect Apple to review the implementation of Object Pascal. It would be nice if objects could be a more orthogonal extension.

Apple's experience with MacApp, and its predecessors on the Lisa indicate that there is

important link-time optimization which is impossible with partial compilations, but that can be easily done with a global analysis of the calling graph at link-time. We need to continue to investigate compiler and linker techniques for efficient implementation of flexible object oriented systems.

And finally, I would like to make a plea to investigate techniques of organizing large numbers of objects, and proving object oriented programs are correct. Most of the spectacular successes of object oriented languages have been in the areas of interactive user interfaces, icon oriented environments, and graphics applications, where it seems that the entire screen often serves as a giant debugger. If objects "misbehave" then it becomes readily apparent from strange things happening on the screen. And using the visual effects of the bug, it is often possible to "hack around" and "get it working right". But allowing objects to save state based on messages, and to have private copies of references to other objects, is an inherently unstructured way of programming, relative to structured programming. Once the efficiency issues are solved, the object oriented world is still likely to need new methodologies of development in order to "scale up" to attack very hard problems.

One promising approach is to organize objects into rather strict hierarchies, and restrict normal message passing to peers, subordinates, and superiors. With such restrictions, it is possible to use a state transition machine to specify most side effects of messages.

### **Acknowledgements**

David Goldsmith directed the investigation and design phases of the C++ project as part of the MacApp project. Cliff Greyson moved the CFront compiler to MPW and made the Apple extensions. Keithen Hayenga and Ron Metzger produced the C++ interfaces to the Macintosh operating system and toolbox. Clayton Lewis directed the numerics work for C++. Joe MacDougald is developing sample programs and a test suite for C++. Don Reed is preparing technical documentation.

I would like to thank David Goldsmith, Clayton Lewis, Roger Lawrence, Herb Kanner, and Joe MacDougald for reading early drafts of this paper and suggesting numerous improvements.

### **References**

1. "AT&T C++ Translator: Release Notes for version 1.2", AT&T Software Sales and Marketing, Greensboro, NC.
2. "MPW 2.0 Reference", Apple Programmers Development Association (APDA), Renton, WA, 1987.
3. "MPW 2.0 Pascal Reference", APDA, Renton, WA, 1987.
4. "Introduction to Object Pascal", Ken Doyle, MacTutor, MacTutor Company, Placentia, CA, Dec 1986. (reprinted by Advanced Technology Group(ATG), Apple Computer, Cupertino, CA.)
5. "MacApp Release 1.1.1", software and documentation, APDA, Renton, WA, 1987
6. "Minimal C++ Report", David Goldsmith and Larry Tesler, Apple Technical Report, ATG, Cupertino, CA, Feb, 1987.

# Extending the C++ Task System for Real-Time Control

Jonathan E. Shopiro

AT&T Bell Laboratories  
Murray Hill, New Jersey 07974

## ABSTRACT

The task system for coroutine programming was one of the first libraries written in C++ and it has served admirably in several applications. It is small, efficient, and easy to use. As part of a robot control project, it was extended to support real-time control. The new task library is more robust, more easily extendible, and more portable than the original. It is upward compatible, so that programs written for the old task library can still be used. This memo documents the new features and the internal structure of the revised system, and is intended for users of the task library, for authors of other coroutine systems, and as a guide to porting the task library to other machine architectures.

## 1. OVERVIEW

The C++<sup>[1]</sup> task library<sup>[2]</sup> is a coroutine<sup>1</sup> support system for C++. A task is an object with an associated coroutine. The task library includes a scheduler that enables each task to execute just when it has work to do, and to wait when necessary for whatever is needed.

Programming with tasks is particularly appropriate for simulations, real-time process control, and other applications which are naturally represented as sets of concurrent activities. A task can represent a simple part of a complex system, and when the task gains control, it can process its current input data, perhaps creating other data that will be processed by other tasks. It can then relinquish control, waiting for more input or an external event.

In a program using the C++ task system, all tasks share the same address space so that pointers can be passed between tasks, and it is easy to share common data structures. Also, the scheduler is non-preemptive, so that each task runs until it explicitly gives up the single processor, and only then does the scheduler choose a new task to run. This eliminates the need for locks on shared data (which would be required if preemptive scheduling or multiple processors were used) and allows task-switching to be accomplished with low overhead, but requires the programmer to be careful that no task monopolizes the processor.

The rest of this section is an overview of control flow in the task system along with a brief note on task system performance. The next section of the paper describes the interrupt handler class and how it can be used to provide real-time response to external events. The following section of the memo, which is primarily intended as a porting guide, gives a detailed explanation of the internals of task switching and creation. Since coroutines cannot be written in portable C or C++, the task library contains non-portable C++ code, and even some assembly code. This memo describes the versions for the VAX,<sup>†</sup> Motorola 68000, and AT&T 3B series computers. Familiarity with C++ is assumed.

1. Coroutines can exchange control among themselves more freely than ordinary functions and procedures. In the usual function calling discipline, when one procedure (more precisely, one instance of a procedure) executes a procedure call, a new instance of the called procedure is created, and the calling procedure waits until the called procedure (and any procedures it may call) returns. A procedure instance is initiated when the procedure is called and is destroyed when it returns. When one coroutine (coroutine instance) initiates another it need not wait for the new coroutine to end, but instead it can be resumed while the new coroutine is still active. A running coroutine can relinquish control to any waiting coroutine without abandoning its state and later regain control and continue from where it left off.

† VAX is a registered trademark of Digital Equipment Corporation.

## 1.1 The Structure of the Task System

Control in the task system is based on a concept of operations which may succeed immediately or be blocked, and objects<sup>2</sup> which may be *ready* or *pending* (not ready). When a task executes a blocking operation on an object that is ready, the operation succeeds immediately and the task continues running, but if the object is pending, the task waits. Control then returns to the scheduler, which chooses the next task from the *run chain*, a list that contains all the tasks that are ready to run (not waiting or terminated). For example, a queue head is ready when the associated queue has data, and *get* (which extracts an item from the queue) is a blocking operation for queue heads. Similarly, *put* is a blocking operation for queue tails, which are ready unless the associated queue is full.

Each different kind of object can have its own way of determining whether it is ready or not, which makes it easy to add new capabilities to the system. On the other hand, each kind of object can have only one criterion for readiness (although it may have several blocking operations), so it is not possible for one object to act as both a queue head and a queue tail, for example.

Each object contains a list (the *remember chain*) of the tasks that are waiting for it. When any operation changes the state of a pending object so that it becomes ready, those tasks are moved to the run chain; this is called an *alert*. Thus the cycle is: a task runs until it blocks; it is saved on the remember chain of one or more pending objects; some other task or an interrupt alerts the object; the original task is moved to the run chain; eventually the task runs again.

## 1.2 Task System Performance

The fundamental operations of the task system are task creation and task switching. In order to make a meaningful evaluation of their performance, equivalent programs using tasks and UNIX<sup>††</sup> Operating System processes were written. These programs are given in the appendix. Each of the first pair of programs (*tcreate.c* and *ucreate.c*) repeatedly creates new trivial tasks (processes) and waits for them to terminate. Each of the second pair of programs (*tswitch.c* and *uswitch.c*) creates a single child task (process) and repeatedly exchanges control with it through a pair of semaphores (see section 2.1.1) in the task version, and through UNIX signals in the process version. The programs were run on a SUN 3/280 under 4.2BSD, using the free store allocator (*malloc.c*) from Ninth Edition UNIX, which is much faster than the one supplied with 4.2BSD. The results were that *tcreate.c* was 37 times faster than *ucreate.c*, and *tswitch.c* was 10 times faster than *uswitch.c*.

It is important to note that the task system and the UNIX Operating System are not equivalent and that the results of these performance measurements do not imply that the task system is 23.5 times better than UNIX. Among the significant differences between tasks and processes are the following.

- A set of tasks runs as a single UNIX process. The task system relies on the UNIX Operating System for I/O, memory management, etc.
- Tasks share an address space, while processes have separate address spaces. This means that tasks can share data by simply passing pointers, while processes must go through one of several much more complex and expensive procedures to share data. By the same token, tasks can interfere with each other as easily as they can cooperate, while errant processes usually kill only themselves.

2. Class `object` is the base class of most classes in the task system. We use the `typewriter` font for programming language constructs.

†† UNIX is a registered trademark of AT&T.

- The task system can support two or three orders of magnitude more concurrent tasks (especially with the `SHARED` option; see section 3.1) than the UNIX Operating System can support processes. It is not uncommon for a simulation to require thousands of tasks.

The memory required for the task system is about 14,000 bytes for code and data, plus about 70 bytes per task, plus stack storage for each task. By default each task has its own stack buffer with a default size of 3000 bytes, but tasks can share a stack buffer and then storage is required only for the active stack of each task (typically 50 to 100 bytes). This option is very useful for applications with thousands of tasks. Queues occupy 60 bytes (including both head and tail) plus the size of whatever is stored on the queue. Lists of tasks are maintained in various places, for example the run chain and remember chains; each occurrence of a task on a list adds 8 bytes to the total memory requirement.

## 2. REAL-TIME EXTENSIONS

The application that motivated this work on the task system was a control system for two robots operating in the same workspace. The most important requirement of this application that was not fulfilled by the original task system was the need for tasks to wait for external events. For example, after a motion command was sent to a robot, the task that sent the command needed to wait for the interrupt that was generated by the robot hardware when the command was complete or had failed. A related requirement of some real time systems is to respond to external events in a timely manner, for example to retrieve data from an unbuffered external device. Also, in the original task system, the scheduler would exit when the run chain was empty. This is inappropriate in a system that is intended to respond to external events because some task might become runnable after an interrupt.

Hardware interrupts are handled differently by different machines and operating systems, so the interface to the task system must also vary. For didactic reasons, the version described here is for the UNIX Operating System using signals as interrupts, but it should be clear how to adapt it to other environments.

In the task system events that can be waited for are represented by instances of class `object` or derived classes. When the function `object::alert()` is called, the tasks that were waiting for that object are made runnable. A natural solution to the problem of waiting for external events was to define a new kind of object to represent external events, and when such an event occurs, to call `object::alert()` for the appropriate object. These objects are called interrupt handlers.

```
class Interrupt_handler : public object {
    int    id;                                // signal or interrupt number
    int    got_interrupt;                     // an interrupt has been received but not alerted
    Interrupt_handler *old; // previous handler for this signal
    virtual void interrupt() {}               // runs at real time
public:
    int    pending();                         // FALSE once after interrupt
    Interrupt_handler(int sig_num);
    ~Interrupt_handler();
};
```

After an interrupt handler is created, a task can wait for it, exactly as for any other object. When the interrupt occurs, the handler's `interrupt()` function will be executed immediately, or rather, as soon as the operating system can route the interrupt to the process. When the interrupt function returns, control will resume at the point where the current task was interrupted.

At the next entry to the scheduler, when the currently running task blocks, a special task, the *interrupt alerter*, will be scheduled. This task alerts the handler (and any other handlers that have received interrupts since it was last scheduled). Thus the waiting task becomes runnable. As long as any interrupt handler exists, the scheduler will wait for an interrupt, rather than

exiting when the run chain is empty. The pending function for an interrupt handler always returns *TRUE* except the first time it is called after an interrupt occurs.

`Interrupt_handler::interrupt()` is a null function, but since it is virtual, the programmer can specify the action to be taken at interrupt time by simply defining an `interrupt()` function in a class derived from `Interrupt_handler`. An example is given in section 2.1.2. In this way real-time response can be obtained without resorting to a preemptive, priority-based scheduler which would be more complex and less efficient, and would require locking of shared data structures.

## 2.1 Avoiding Interference

Whenever shared data structures are manipulated by concurrent processes, there is the potential for interference, where one process is in the middle of modifying a data structure and another process accesses it and finds it in an invalid state. Segments of code that access shared data structures are called *critical regions*<sup>[3]</sup>. If more than one process can be in a critical region at one time, there is a potential for interference.

Interference is easy to avoid in the task system, because of the non-preemptive nature of the scheduler. There are only two ways in which interference can arise: a task switch occurring within a critical region, or an interrupt routine that accesses shared data.

It is almost always possible to write code so that no operation that could cause a task to block is inside a critical region. The style of programming where coroutines share information by sending messages to each other in the form of objects on queues typically leads to programs where there are no shared data structures or critical regions at all. Even if coroutines must share access to a data structure and alternately modify it, no problems will arise if the routines that do the modification refrain from operations that could cause the task to block. A properly modular program will generally satisfy this requirement without any extra effort.

**2.1.1 Semaphores** If, for some unusual reason, it is necessary to put an operation that could cause the task to block in a critical region, then the affected data structure should be protected by a semaphore, which will allow only one task at a time to access the object. The following example code outlines this technique.

```
class My_data {
    Semaphore    sema;
    // user data
public:
    void lock() { sema.wait(); }
    void unlock() { sema.signal(); }
    My_data() : sema(1) { ... } // see note3
};
```

Each critical region must begin with a call to `My_data::lock()` for the object to be accessed, and end with a call to `My_data::unlock()`. This will ensure that no interference occurs, even if the operations in the critical region cause the task to block.<sup>4</sup>

The implementation of semaphores using the task system is easy.

3. Semaphores which are used for mutual exclusion are initialized with one excess signal so that the first lock call will succeed.

4. But watch out for deadlock.

```

class Semaphore : public object {
    int    sigs; // the number of excess signals
public:
    Semaphore(int i =0) { sigs = i; }
    int    pending() { return sigs <= 0; }
    void    wait();
    void    signal() { if (sigs++ == 0) alert(); }
};

void
Semaphore::wait()
{
    for (;;) {
        if (--sigs >= 0)
            return;
        sigs++;
        thistask->sleep(this);
    }
}

```

Semaphores are useful tools for building other kinds of synchronization besides mutual exclusion. For example, whenever one task wants to wait for an operation to be completed by another task, it can wait on a semaphore.

**2.1.2 Interrupts** The other case where interference can occur is a little more complex. The `interrupt()` routine of an `Interrupt_handler` can be executed at any time, and it would be contrary to the reason for its existence to lock it out. The mechanism that alerts the handler after the interrupt has occurred is carefully designed to be safe from interference, and sometimes the alert is all that is necessary for an application. If it is necessary to gather data from an external device immediately after an interrupt occurs, but the interrupts do not come in rapid succession (for example, the next interrupt won't occur until after the device is reset), the interrupt routine can save the data and the task that is waiting for the interrupt can process the data before resetting the device. In this case even though the data is shared, the interrupt routine cannot access the data at the same time as the task.

Sometimes, however, it is necessary to handle interrupts that can come in rapid succession, with a requirement to gather data at each interrupt, so that several interrupts may occur before the task that will process the data can be scheduled, and more interrupts may occur even while the task is running. This problem is best handled by establishing a queue of the interrupt data records. Then the only shared data between the interrupt handler and the task processing the data can be the queue head and tail pointers, which can be atomically updated. In the following toy example, the interrupt routine records the value returned by an arbitrary function, `get_data()`, each time the signal `SIGINT` is sent. A waiting task is then scheduled and prints all accumulated data.

```

class Delete_handler : public Interrupt_handler {
    void    interrupt();
    int*    localq;        // data buffer beginning
    int*    localq_end;    // data buffer end
    int*    localq_h;      // queue head
    int*    localq_t;      // queue tail
public:
    int     getX(int&);    // the next item, if any
    Delete_handler(unsigned local_q_size =5);
    ~Delete_handler() { delete [localq_end - localq] localq; }
};

```

The delete handler (so called because `SIGINT` is normally sent when the user presses the delete key) is an interrupt handler that maintains a local queue of data. Its interrupt function will put data on the local queue, using `localq_t`, the queue tail pointer, and its `getX()`

function is used by a task to retrieve the data.

```
Delete_handler::Delete_handler(unsigned local_q_size)
: (SIGINT)    // base class constructor arg
{
    localq_t = localq_h = localq = new int[local_q_size];
    localq_end = &localq[local_q_size];
}
```

The constructor initializes the local queue. The size of the local queue determines how many interrupts can be awaiting processing.

```
void
Delete_handler::interrupt()
{
    register int*p = localq_t;
    *p = get_data();
    if (++p == localq_end) p = localq;
    if (p != localq_h)
        localq_t = p; // no overflow
    else error("Overflow");
}
```

The interrupt function assumes that `localq_t` points to an available slot in the queue and puts the real-time data there. It then checks for overflow and updates `localq_t` to point to the next available slot if it's okay or calls an error function otherwise.

```
int
Delete_handler::getX(int& ans)
{
    register int*p = localq_h;
    if (p == localq_t)
        return 0;
    ans = *p;
    if (++p == localq_end) p = localq;
    localq_h = p;
    return 1;
}
```

The function `getX()` assigns the next datum to its argument and returns 1, or returns 0 and leaves its argument alone if no data is available. A call to `getX()` may be interrupted, but it has been designed so that no corruption of the queue will result.

```
class Delete_printer : public task {
    Delete_handler*    handler;
public:
    Delete_printer();
};
```

`Delete_printer` is a task that will create a `Delete_handler` and print whatever data is received.

```
Delete_printer::Delete_printer()
: handler(new Delete_handler)
{
    for (;;) {
        wait(handler);
        int    i;
        while (handler->getX(i))
            cout << i << "0;
    }
}
```

Note that each time the printer task is scheduled, it prints all the available data from the delete handler.

## 2.2 Implementation Details

The approach taken was to minimize the impact to the scheduler and to isolate as much as possible the machine and operating system dependent parts of the implementation. There is a system-dependent function, `sigFunc()`, which catches each signal for which an `Interrupt_handler` exists. When the signal is sent, `sigFunc()` calls the appropriate `interrupt()` function. It then atomically puts the address of a dedicated alerter task in a static, private cell of the scheduler and rearms the signal and returns. At the next entry to the scheduler, that cell is checked and if it is non-zero, the alerter task is scheduled. The alerter task alerts all pending interrupt handlers and returns to the scheduler. Tasks that were waiting for interrupt handlers are then eligible to run.

The other system-dependent parts of the implementation are the constructor and destructor for class `Interrupt_handler`. Its constructor takes the signal number as argument (it might be an interrupt vector address in another system). If some other interrupt handler already existed for that signal, it is saved (and alerted if it was pending), and otherwise the UNIX system function `signal()` is called to associate `sigFunc()` with the signal. The destructor undoes the action of the constructor, restoring the previous signal routine if necessary.

## 3. TASK SYSTEM INTERNALS

The primary task system datatypes that are visible to the programmer are shown in figure 1.

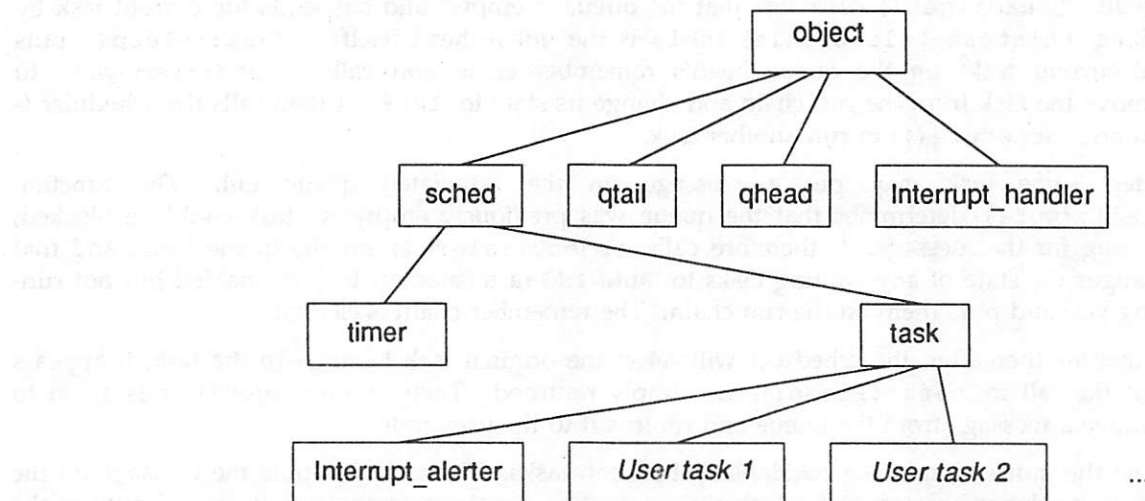


Figure 1. Classes in the Task System

Coroutines are represented by objects derived from class `task`. The executable code of the coroutine is the constructor of the derived class.

The basic function of a coroutine system is, when one task is to pause and let another task resume execution, to save the state of the old task in some suitable data structure, and to restore that of the new task. The relevant components of the state are the machine registers and the stack. The C++ task library does this in a special way: it uses the ordinary function call sequence to save the machine registers in an ordinary stack frame and then saves just the frame pointer register (and on the 3B, the argument pointer) in the task data structure, and it uses the ordinary function return sequence (after resetting the frame pointer) to resume execution of the next task. This minimizes machine dependencies.

### 3.1 Task Switching

The central data structure of the task system is the list of runnable coroutines, pointed to by `sched::run_chain`. Whenever the scheduler needs to choose a new task to run, it selects (and removes) the first task<sup>5</sup> on the run chain. When the running task blocks, it calls `task::sleep()` with the pending object as argument, which puts the task on the object's remember chain and calls the scheduler to run another task. When the object is no longer pending, the function `object::alert()` is called, and all the tasks on that object's remember chain are moved to the end of the run chain. This scheduling discipline is called first come, first served (FCFS).

**3.1.1 Blocking** Figure 2 shows the essential part of the typical code executed when the running task blocks.

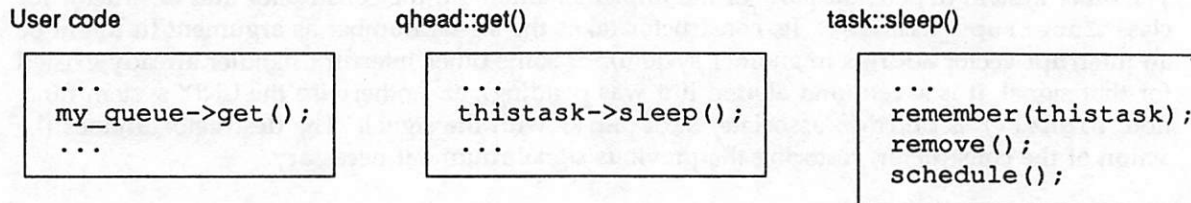


Figure 2. Task Blocking

Here the user code calls the queue head's `get()` function to retrieve the next message from the queue. `Qhead::get()` discovers that the queue is empty<sup>6</sup> and suspends the current task by calling `this->sleep(this)` (this is the queue head itself). `Task::sleep()` puts the current task<sup>7</sup> on the queue head's remember chain and calls `sched::remove()` to remove the task from the run chain and change its state to `IDLE`. It then calls the scheduler (`sched::schedule()`) to run another task.

Later, some task may put a message on the associated queue tail. The function `qtail::put()` determines that the queue was previously empty, so task could be blocked, waiting for the message. It therefore calls `object::alert()` for the queue head, and that changes the state of any waiting tasks to `RUNNING` (a misnomer: they're enabled but not running yet) and puts them on the run chain. The remember chain is cleared.

Sometime thereafter, the scheduler will select the original task to run. To the task, it appears that the call to `task::sleep()` has simply returned. Then `qhead::get()` tries again to remove a message from the queue and returns it to the user code.

Note the indirection of the relationship between tasks. The task that puts the message on the queue need not be aware of which task is waiting for the message and its act of putting the message on the queue does not run the waiting task, but rather makes it runnable. This is an indication of the flexibility of task-based programming.

The central routine for task switching is `sched::schedule()`. It is called when the current task is blocked and another task must be run. It is most commonly called by `task::sleep()`, as above, when the currently running task is waiting for a pending object, but it can also be called by

5. If the run chain is empty, all tasks are sleeping. If there are any interrupt handlers, the scheduler waits for an interrupt (see *REAL-TIME EXTENSIONS*), but otherwise it exits since no task could ever become runnable.

6. If the queue is not empty, `qhead::get()` simply returns the next message to its caller.

7. The macro `this` returns the value of a private static data item, which is the current running task.

- `Task::~~task()`, when the currently running task is destroyed;
- `Task::resultis()`, when the currently running task finishes;
- `Task::delay()`, when the currently running task waits for a simulated time;
- `Timer::resume()`, when a (simulated time) timer times out.

`Task::sleep()`, in turn, is called whenever a task executes a blocking operation on a pending object. In addition to `qhead::get()`, which blocks when it tries to get an object from an empty queue as described above, the other blocking operations in the supplied classes are

- `Qtail::put(object*)` blocks when it tries to put an object onto a full queue;
- `Sched::result()` blocks when it tries to retrieve the result of a task that hasn't finished;
- `Task::wait()`, `task::waitlist()`, or `task::waitvec()` block when they are called with arguments<sup>8</sup> that are still pending.

3.1.2 *Resuming* Figure 3 shows the main pieces of code that are used to resume a coroutine.

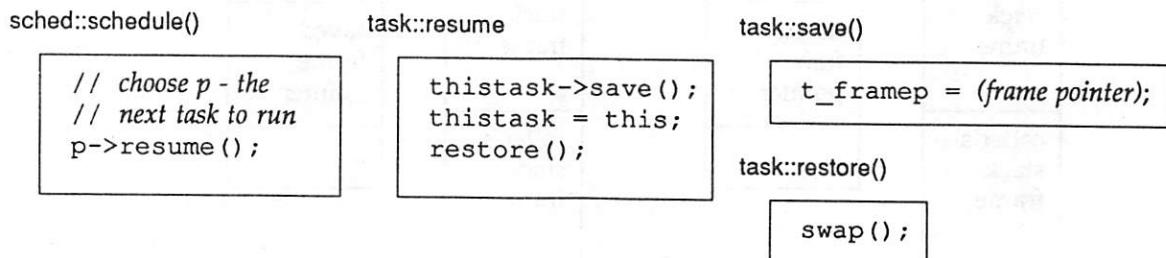


Figure 3. Resuming a task

The primary function of `sched::schedule()` is to choose the next task<sup>9</sup> to run, and then to call the `task::resume()` function for the chosen task. `Task::resume()` then calls `task::save()` for the current task, to save enough information to resume it later, changes the current task pointer (`object::thistask`) to point to the new task, and finally calls `task::restore()` for the new task. `Task::restore()` calls an assembly language<sup>10</sup> routine `swap()` and there the magic occurs; when `swap()` returns, the new task is running.

In order to understand the trickery of the task switch, a little information about the C runtime environment and stack<sup>[4]</sup> is required. All the temporary information associated with a C function is stored in the hardware registers and a data structure called a *stack frame*.<sup>11</sup> As each function is called and entered, a new stack frame is created on top of the stack. When the function returns, its stack frame is popped, and the caller's stack frame is made current. There is a hardware register called the *frame pointer*, which defines the current frame: every access to the current frame goes through the frame pointer. The frame contains at least the arguments<sup>12</sup> to

8. When `task::waitlist()` or `task::waitvec()` is called with more than one object argument, the task is put on the remember chain of each argument using `object::remember()` and later removed using `object::forget()`.

9. A simulated time timer can also be scheduled, but the mechanism described here isn't used. The timer simply makes any tasks waiting for it ready (by calling `object::alert()`) and re-enters the scheduler. The real-time timer given in the appendix does use this mechanism.

10. This is the only assembly language in the system. The VAX version is four instructions long, and the MC68000 and 3B versions are each six instructions.

11. This description applies to the MC68000, VAX, and 3B UNIX C implementations. Other C implementations may do things differently. Good luck.

12. On the 3B, although the arguments are part of the frame, they are accessed through another register, the *argument pointer*.

the function, the saved registers, the return address, and the location of the caller's stack frame.

Figure 4 shows the primary data structures used in task switching. The objects above the dashed line refer to the current task, and those below are repeated for each task in the system.

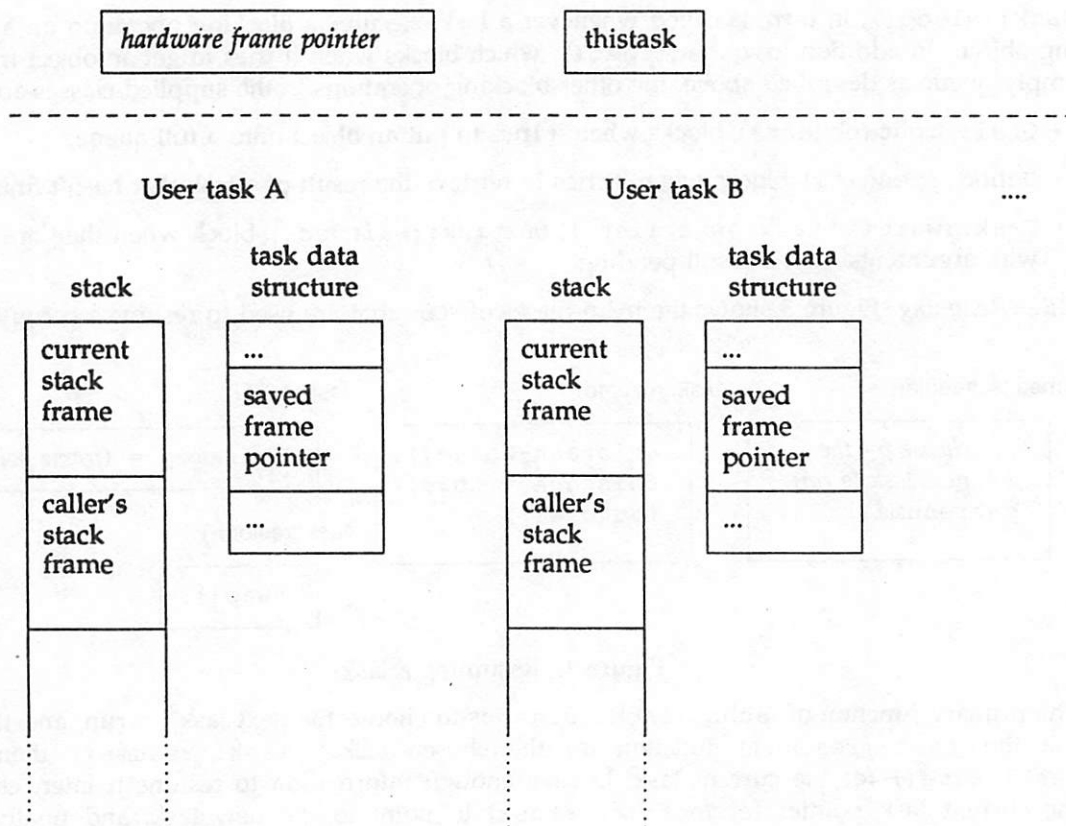


Figure 4. Task system data objects

Each task has its own stack, and when the task is running there must be memory available so that its stack can grow as necessary. This memory is called a *stack buffer* and may be either the original UNIX stack segment, or a large block of memory allocated from the free store. Normally<sup>13</sup> each task has its own dedicated stack buffer, and its stack is always stored there, whether or not the task is running. Alternatively, several tasks can share a buffer, and then at any time the stack belonging to one of the tasks is occupying the buffer, and the stacks of the others are in temporary storage areas allocated from the free store. Before running a task, if its stack is not already in its stack buffer, the stack currently in the buffer is copied out to a newly allocated area of memory, and the stack belonging to the task that is about to run is copied in (and its temporary storage area reclaimed). Sharing stack buffers can lead to substantial savings in memory for applications with many tasks, since stack buffers are typically an order of magnitude larger than the stacks they contain.

The task data structure contains the location of the stack buffer, whether it is dedicated or shared, the location of the stack when it is not in the stack buffer, and a variety of other such information.

13. The default mode argument to the task constructor is `DEDICATED`, and the default value for the `stacksize` argument is 3000 (bytes). If the mode argument is `SHARED`, the `stacksize` argument is ignored.

C functions use registers both for programmer-declared register variables and as temporaries. Small (one or two word) return values are usually passed in registers, too. The registers are divided into two sets: those which can be modified freely within a function, and those which must be preserved or restored. The former, volatile, set is primarily used for temporaries and function return values, and the latter is used for register variables. On the VAX, the non-volatile registers are *R2-R11*, on the MC68000, *d2-d7* and *a2-a5*, and on the 3B2, *R3-R8*. When a C function regains control after a function call (i.e., when the called function returns), it expects that the values of the non-volatile registers will not have changed. This is achieved by having the called function save on entry the ones it plans to modify, and restore them on return.

Temporaries and function arguments are pushed onto the stack using another register called the *stack pointer*. The stack pointer is always at the top of the stack (by definition — the top of the stack is where the stack pointer is) and the frame pointer is usually somewhat below<sup>14</sup> it. The function return sequence uses the frame pointer to access the current stack frame. The return sequence restores all the registers that were saved on function entry, resets the frame pointer to point to the caller's stack frame, adjusts the stack pointer, and continues execution at the next instruction of the caller. It is thus ideal for task switching. *Resuming a task is done by resetting the frame pointer to the address of the frame saved when that task was suspended, and executing the return sequence.* This is the job of `swap()`.

Since a coroutine always gives up control through a function call, and regains control through a return, the task system need not save or restore the volatile registers. On the other hand, when a coroutine regains control, not all of the functions that have been called since it lost control have returned, and therefore the task system must make a special effort to restore the non-volatile registers. This is arranged by declaring and initializing<sup>15</sup> a sufficient number of register variables in the code of those task system functions that are called when a coroutine gives up control. There are only two such functions: `task::task()` which is called to create a new task (which then gains control), and `sched::schedule()` which handles all other cases. It is necessary to examine the generated assembly code (both with and without optimization), to determine that the appropriate registers are being saved.

**3.1.3 The nitty-gritty** The function `task::save()` looks into the current stack, back two frames, to find the address of the frame associated with `sched::schedule()` and stores it in `task::t_framep`. The procedure for doing this varies from machine to machine depending on the stack frame layout. The differences between machines are encapsulated in macros defined at the beginning of `sched.c`. It is crucial that `task::save()` is only called from `task::resume()`, which in turn is only called by `sched::schedule()`, so that `task::save()` knows how far to go back in the stack. Also, `task::save()` checks for stack overflow, which is a non-recoverable error. If the task is shared, `task::save()` also calculates the size of the stack and saves it in `task::t_size`.

`Task::restore()` is then called for the new task. If the new task's stack is already in the appropriate stack buffer (always the case for `DEDICATED` tasks), it calls `swap()`, which simply resets the hardware frame pointer to the saved address of the `sched::schedule()` frame and returns. Otherwise, the stack currently occupying that buffer (which may not belong to the task that was just saved) is copied out to newly allocated space, and `copy_in()` is called to copy the new stack into the buffer.<sup>16</sup> This is done with some care to avoid over-

14. Don't be confused by the fact that in both the VAX and MC68000, the stack grows toward lower addresses, so that the top of the stack has the lowest address of any stack element, and the value of the stack pointer is always less than the value of the frame pointer. On the 3B2, it's just the opposite.

15. If you don't initialize them the compiler will optimize them away.

16. This copy out, copy in sequence only occurs when a shared task is about to be executed and one of the other tasks sharing the same stack buffer ran more recently than the new task. Thus space and time can be optimized by creating tasks that will alternate execution in different stack buffers.

writing the current frame. `Copy_in()` then calls `swap()`, which returns to the new task.

On the VAX, the C return sequence is a single instruction: `RET`. The stack frame is completely self-describing, and the `RET` instruction is able to figure out what registers were saved in the frame and restore them properly, and do everything else that is necessary to return from a function. In other words, you can return from any function by setting the frame pointer to the frame saved by that function, and executing the `RET` instruction. Thus `swap()` for the VAX is particularly easy.

On the MC68000, things are not so simple. The stack frame is not self-describing; you cannot tell what registers are saved or where by looking at the stack frame itself. The return sequence of a C function consists of several instructions which depend on the structure of that function's stack frame. This is significant because in order for `swap()` to correctly hand control to the new task, it has to return to the caller of `sched::schedule()`, or in the case of a new task (described below), `task::task()`. This requires that these two functions have frames with identical structure, so that their return sequences match, and that this return sequence is copied into `swap()`. This can only be verified by examining the assembly language output of the C compiler.

On the 3B2, things are complex in a different way. There are two registers that point to different parts of the frame, the frame pointer, and the argument pointer. The function return sequence uses the frame pointer to restore the saved registers, and the argument pointer to reset the stack pointer. This requires that both the frame pointer and the argument pointer be saved in the task data structure and reset by `swap()`. Also the return sequence must specify that six registers were saved (but not which individual registers, as in the MC68000).

### 3.2 Creating New Tasks

In the C++ task system, the coroutines are written as constructors of classes which are directly derived from the supplied class, `task`, as in the following example.

```
struct My_task : task {
    My_task();    // constructor
    // other function and data members
};
My_task::My_task() // the coroutine "main" function
{
    // coroutine body
}
```

When an object of such a class is created by the C++ operator `new`, a new task is initialized and the constructor executes as its body. An example of the code that is written by the programmer in the old task to create a new task is the following.

```
{
    // ...
    My_task*    p = new My_task;
    // ...
}
```

This suspends the current task and puts it on the run chain (so that it can be resumed as soon as any task blocks), and begins execution of the new task.

The function that is responsible for creating the new thread of control is `task::task()`, the constructor of class `task`. This arrangement falls naturally out of the normal C++ object creation sequence: the first step in initialization of an object is to initialize that part of the object that corresponds to the base class of the class of the object. The strategy for creating a new thread of control is to duplicate the current stack, and modify one of the copies so that `task::restore()` will resume the parent, while in the other copy it will resume the child.

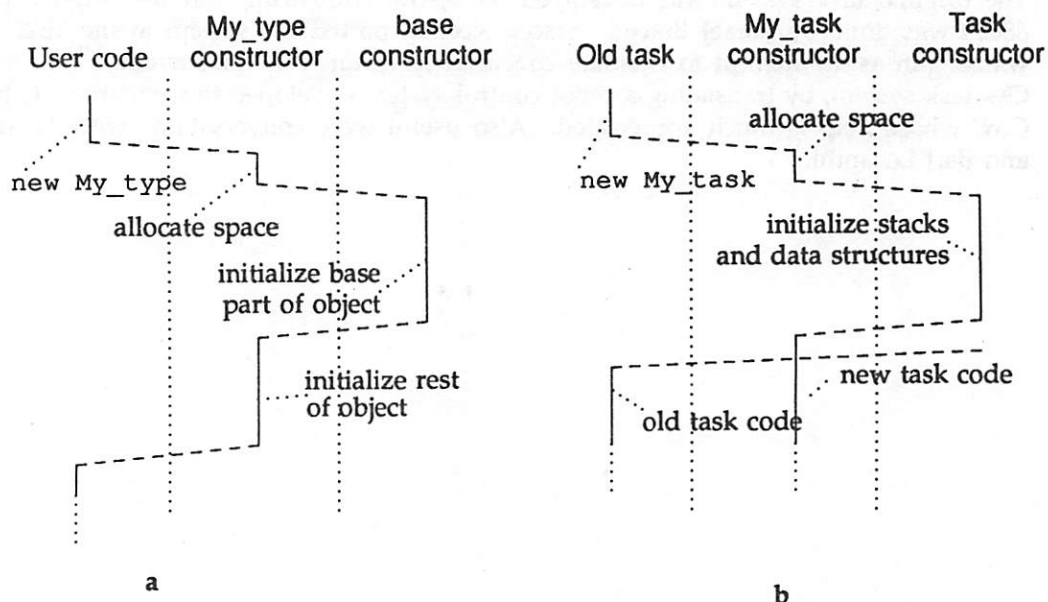


Figure 5. Object construction vs. task initialization

Figure 5 shows the flow of control as an ordinary object is created (a), compared to the flow of control in the creation of a new task (b). In both, a new expression initiates the operation. This calls the appropriate constructor of the named class (`My_type` or `My_task` in the figure). Before the first programmer-written action, the constructor allocates space for the object, and then calls the constructor of the base class. In the case of task creation, this is `task::task()`. At this point the frame at the top of the stack is that of `task::task()`, the frame under it is that of the new task constructor (`My_task::My_task()` in our example), and the frame below that belongs to the old task. Thus, if `task::task()` were to simply return, the new task constructor, which is the body of the new task, would run, but there would be no way to resume the parent task, so the job of `task::task()` is to provide a way to resume the parent. It does this by saving the frame pointer of the new task frame (`My_task::My_task()`) in the task data structure of the old task, so that the restore sequence described above will resume the old task.

`Task::task()` then initializes the task data structure<sup>17</sup> and copies the current stack so that a return sequence can be executed in both the original stack and the copy. If the new task is dedicated, that is, it will have its own stack buffer, the new buffer for the child is allocated from the free store and the stack is copied directly into it. Otherwise the new task is to share the stack buffer of its parent, and the parent's stack is copied<sup>18</sup> to newly allocated space. After the stack is copied, the function `task::fudge()`<sup>19</sup> is then called to modify the `My_task::My_task()` stack frame, in the parent's copy of the stack, so that the return sequence will fully restore the parent task. At a minimum, the complete set of saved registers from the `task::task()` stack frame are copied into the parent stack frame. After the call to `task::fudge()` has prepared the parent copy of the stack, `task::task()` returns in the child's copy of the stack, and the new task is running.

17. In case this is the first call to `task::task()`, an additional task data structure is created for the main program.

18. The parent's stack will be copied back into its original position before it runs again. A task's stack is always in the same place when it runs, although it may be copied out and back in again while the task isn't running.

19. Stack frame modification is highly machine and compiler dependent: `task::fudge()` is appropriately named.

#### 4. ACKNOWLEDGEMENTS

The original task system was developed by Bjarne Stroustrup, and the original port to the 68000 was done by Rafael Bracho. Stacey Keenan ported the system to the 3B2. This work was begun as an attempt to examine concurrency features of Concurrent C<sup>[5]</sup> to those of the C++ task system, by translating a robot control system developed in Concurrent C by Ingemar Cox, whose help is much appreciated. Also useful were conversations with Thomas Cargill and Bart Locanthi.

#### REFERENCES

1. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.
2. Bjarne Stroustrup. A set of C++ classes for coroutine style programming. AT&T Bell Laboratories Computing Science Research Center CSTR-90 (July 1982, revised November 1984).
3. Per Brinch Hansen. *Operating System Principles* Prentice-Hall, 1973.
4. Johnson, S. C., and Ritchie, D. M. The C language calling sequence. AT&T Bell Laboratories Computing Science Research Center CSTR-102 (1980).
5. N. H. Gehani and W. D. Roome, "Concurrent C", *Software-Practice and Experience*, vol 19:9, (Sept. 1986) 821-844.

## EXAMPLE PROGRAMS

### *tcreate.c*

The following program repeatedly creates a task and waits for it to terminate. It would be possible to time creation of new tasks without waiting for them to terminate, but because of the limited number of processes that can exist under UNIX, the corresponding UNIX program would fail.

```
#include "task.h"

class Child : public task    // user task declaration
{
public:
    Child(int); // task constructor declaration
};

Child::Child(int i) // user task constructor definition
: ("Child")         // argument to base class constructor
{
    resultis(i);    // terminate task execution
}

main()
{
    for (register int i = 10000; i--; ) {
        Child* c = new Child(i); // create a task
        c->result();              // wait for it to terminate
        delete c;                // clean up
    }
    thistask->resultis(0); // exit from main task
}
```

### 2.1 *ucreate.c*

The following C program repeatedly forks a UNIX process and waits for it to terminate.

```
main()
{
    register int i;
    for (i=10000; i--; )
        if (fork() == 0)
            exit(0); // child process
        else
            wait((int*)0); // parent process
}
```

2.2 *tswitch.c*

The following program uses two semaphores (described in section 2.1.1) to alternate control between a parent and child task.

```
#define K 10000
#include "task.h"

class Child : public task
{
public:
    Child();
};

Semaphore sema1;    // for signals from main to Child
Semaphore sema2;    // for signals from Child to main

Child::Child()
: ("Child")
{
    for (register int n = K / 2; n--; ) {
        sema1.wait();    // wait for a signal from main
        sema2.signal();  // send it back
    }
    resultis(0);
}

main()
{
    new Child;
    sema1.signal();      // send the first signal
    for (register int n = K / 2; n--; ) {
        sema2.wait();    // wait for a signal from Child
        sema1.signal();  // send it back
    }
    thistask->resultis(0);
}
```

2.3 *uswitch.c*

The following C program uses a UNIX signal to force alternation between two UNIX processes. The program is a little strange in that its main routine consists of an infinite loop of `pause()` calls. Unfortunately the utility of `wait()` and `pause()` for signal handling is limited because it is always possible that a signal has been received just as the `wait()` or `pause()` is being executed.

```
#include <signal.h>
#define K    10000
int  otherpid;
int  received;
int  child;
void
sig() /* signal-catching routine.  called when a signal is received */
{
    signal(SIGTERM, sig); /* arrange to catch the next signal */
    received++;
    if (child && received >= K/2) exit();
    kill(otherpid, SIGTERM); /* send it back */
    if (!child && received >= K/2) exit();
}

main()
{
    signal(SIGTERM, sig); /* arrange to catch the signal */
    if ((otherpid = fork()) == 0) { /* create the child process */
        otherpid = getpid(); /* get parent process id */
        child = 1; /* this is the child */
        kill(otherpid, SIGTERM); /* send the first signal */
    }
    for(;;)
        pause();
}
```

2.4 *real\_timer.c*

In addition to the robot application, the system was implemented on the UNIX Operating System using signals as interrupts. A class `Real_timer`, modelled on the original class `timer` was built.

```
class Real_timer : public object {
    friend class Alarm_handler;
    int  state; /* RUNNING, IDLE, TERMINATED */
    long time; /* initially delay, then alarm time */
    void insert(int); /* put on chain */
    void remove(); /* remove from chain & make IDLE */
    void resume(); /* called when time is up */
public:
    Real_timer(int);
    ~Real_timer();
    int pending();
    void reset(int);
    void print(int, int = 0);
};
```

Instead of simulated clock ticks, class `Real_timer` measures time in seconds. It is based on the following handler for the alarm signal and a task that maintains the list of unexpired `Real_timer` instances.

```

class Alarm_handler : public task {
    friend Real_timer;
    Real_timer* chain;
    Interrupt_handler* bell;
    void add_timer(Real_timer*);
    void remove_timer(Real_timer*);
public:
    Alarm_handler();
};

Alarm_handler alarm_handler; // the only instance

Alarm_handler::Alarm_handler()
: ("Alarm_handler"), chain(0)
{
    sleep();
    for(;;) {
        for (long now = time(0); chain && chain->time <= now;
             chain = (Real_timer*)chain->o_next)
            chain->resume(); // alert the timer
        if (chain) {
            alarm(chain->time - now);
            wait(bell);
        } else {
            bell->forget(thistask);
            delete bell;
            sleep();
        }
    }
}

```

The Interrupt\_handler pointed to by Alarm\_handler::bell only exists while there are pending Real\_timer objects. The Alarm\_handler task runs after an alarm signal, and after alerting any timers that have expired, if there are any unexpired timers, it resets the alarm and waits.

# C++ on a Parallel Machine

Thomas W. Doeppner Jr.<sup>1</sup>

Alan J. Gebele<sup>2</sup>

Department of Computer Science

Brown University

Providence, RI 02912

## 1. Introduction

Exploiting a computer with multiple processors can often be difficult because suitable programming-language support for writing parallel programs does not exist. We have integrated C++ with a system called Threads (described below), enabling C++ to take full advantage of parallelism on a parallel computer such as the Encore Multimax<sup>3</sup>. This was accomplished by adding a set of *classes*, the *tasking package*, to the C++ library which the programmer can use to create and synchronize concurrent activity. These classes are based on the system described in [Stroustrup 2], which was designed for a single processor and used a coroutine linkage between concurrent activities (or *tasks*). While extending most of the functionality of Stroustrup's system to a true parallel machine, we have added a class *monitor* to take full advantage of the facilities provided by Threads.

## 2. Overview

The tasking package is based on four C++ classes to be used by the programmer as base classes to build their own classes for parallel computations (see [Stroustrup 1]). The first of these parallel classes is the class *task*. The constructors of any class derived from *task* run as a separate thread of control in the program. The second class, *monitor*, defines a set of routines to make a derived class act as a monitor [Hoare] for the controlled sharing of data between different tasks. A third concept, based on the two classes *qhead* and *qtail*, defines a one-way queue for communication between tasks. *TASKenvironment*, which is not defined directly by the user, is used to set and maintain task-system environment defaults for creating tasks and preemptive scheduling.

### 2.1. The Task Class

The base type *task* creates a separate thread of control for execution of part of a program. Normally, in order to define a task, one defines a class that is derived from *task* which contains only the constructor. It may look like:

```
struct scout : task {
    scout::scout(char*);
};

scout::scout(char *s)
{
    int i = 0;
    while (*s) if (*s++ == ' ') i++;
    resultis(i);
}
```

<sup>1</sup> Doeppner's work was supported in part by a grant from the Encore Computer Corporation, by the Office of Naval Research and Defense Advanced Research Projects Agency under contract N00014-83-K-0146 and ARPA Order No. 4786, and by National Science Foundation Grant MCS8121806.

<sup>2</sup> Gebele is currently affiliated with Bellcore and was supported by Bellcore for this work.

<sup>3</sup> Encore and Multimax are trademarks of the Encore Computer Corporation.

This task counts the number of spaces in an input string and is executed in parallel with anything else being done by the program. The routine *resultis()* is used to set a return value for the task so that some other task may get the value. *resultis()* also tells the tasking package to end the execution of the task.

To activate the task, we simply declare an instance of the class *scount* to create a new thread of control:

```
main(int argc, char *argv[], char *envp[])
{
    scount s("a line with four spaces");

    // do some other work.....

    printf("result is %d\n", s.result());
}
```

The routine *result()* returns the value from the call to *resultis()* but only after the task has completed. Tasks can be in three states, RUNNING, IDLE, or TERMINATED. Normally, a task is in RUNNING state until it exits either by returning through the normal execution or by a call to *resultis()*, when it becomes TERMINATED. A task is IDLE if it stops execution for some reason; for instance, if it calls *result()* when the task from which it is asking for a value has not yet completed. If a task returns without calling *resultis()*, the value of *result()* is -1.

The *task* class gives other services for synchronization of execution in addition to waiting upon the termination of a task. A task may wait upon another task by calling the *wait()* operation. The waiting task goes to an IDLE state until the other task either calls a *wakeup()* operation, or completes execution and goes to a TERMINATED state. The *wait()* and *wakeup()* operations are based on wait queues associated with a task. Each task has one wait queue by default, whose number is 0; at creation time, a task may allocate more queues by passing a parameter to the task constructor. If no argument is given, calls to *wakeup()* and *wait()* act on queue 0. To specify another queue, give its number as the argument.

In the following example, the *wait()* operation is used to ensure that all the threads doing computations are completed before any results are printed. This example multiplies two matrices by computing each member with a separate task. Note that here we do not use the *wakeup()* operation, only *wait()*. Hence the synchronization waits for the task to be TERMINATED. (This use of tasking for matrix multiplication is strictly an illustrative example; even though our implementation of tasking is very inexpensive, it is not cheap enough to make the following example technique practical. However, we are currently developing an even cheaper form of tasking, discussed below, which would be practical for this example.)

```
class matrix {
    int *vec ;
    int len ;
public:
    matrix(int, int);
    ~matrix() { delete vec ; }
    int get(int r, int c) { return vec[len*r+c] ; }
    void put(int r, int c, int val) { vec[len*r+c] = val ; }
    int length() { return len ; }
};

matrix::matrix(int l, int initval = 0)
```

```

{
    len = 1;
    vec = new int[1*1];
    for (int i = 0; i < 1*1; vec[i++] = initval);
}

#define MSIZE 3

matrix A(MSIZE);
matrix B(MSIZE,8);
matrix C(MSIZE);

struct mult : task {
    mult(int, int);
};

mult::mult(int r, int c) : ("mult")
{
    int t = 0;
    for (int i = 0; i < A.length(); i++)
        t += A.get(r,i) * B.get(i,c);

    C.put(r,c,t);
}

main(int argc, char *argv[], char *envp[])
{
    // declare array of mult pointers
    mult *mvec[MSIZE*MSIZE];

    // initialize A
    for (int r = 0; r < MSIZE; r++)
        for (int c = 0; c < MSIZE; c++)
            A.put(r,c,(r+1)*(c+1));

    // startup mult tasks
    for (r = 0; r < MSIZE; r++)
        for (c = 0; c < MSIZE; c++)
            mvec[r*c] = new mult(r,c);

    // wait on all threads to complete
    for (r = 0; r < MSIZE; r++)
        for (c = 0; c < MSIZE; c++)
            mvec[r*c]->wait();

    printf("the result is :\n");
    for (r = 0; r < MSIZE; r++)
        for (c = 0; c < MSIZE; c++)
            printf("c[%d,%d] = %d\n", r, c, C.get(r,c));
}

```

Several other useful operations exist within the *task* class. First, the *wakeupall()* operation wakes up all tasks waiting on a queue (not just the first task, as *wait()* does).

There are also several operations for checking the state of a task which are often useful in debugging: *print()* prints to stdout a summary of the state of the task, *name()* returns a pointer to the string containing the name given the task at creation time, *status()* returns the running status of the task, and *priority()* returns the current priority of execution the task. The package also provides a method of changing the task's priority by calling *setpriority()*.

All computation in the program, even in the *main()* routine, is done in the context of some task. At the beginning of execution, the program initiates the Threads system and then starts up the first task, the *main()* routine. Actually, at compile time the name of *main()* is changed to *TASKmain::TASKmain()*, which is really a class constructor derived from *task*. Hence the declaration line of *main()* can also be written as:

```
TASKmain::TASKmain(int argc, char *argv[], char *envp) :("TASKmain")
```

This means that the *TASKmain()* can be scheduled and waited upon and can give results just like any other task. (Because the tasking package redefines *main()*, all three arguments to *main()* must be included, even though the program does not necessarily use any of them.)

## 2.2. The Monitor Class

The class *monitor* is designed to help the programmer build classes that act like traditional monitors for data access synchronization. Since C++ does not give syntactic support for monitors, a few routines are included to make the construction of monitors possible. Each operation in the derived class must first call *enter()* to acquire the monitor, and must call either *exit()* or *signal()* to relinquish the monitor. The monitor can have any number of conditions associated with it. In calling the constructor for the monitor, the second argument tells how many conditions to create with the monitor; the default is 1.

The monitor has the usual condition operations, *wait()* and *signal()*, which have the standard interpretations (note that *signal()* also releases the monitor). In addition to those calls, the tasking package also supports two other condition operations. First, *signalandwait()* allows the programmer to signal on one condition and wait immediately on another condition without having to reacquire the monitor. Second, *waitevent()* is an application of the UNIX system call *select* with the *wait()* operation which allows the task to synchronize on external events along with monitor conditions.

As an example, we give a solution to the producer/consumer problem. In this standard synchronization problem, two types of tasks, *producers* and *consumers*, communicate by using a common buffer (of finite size). A producer may attempt to put data into the buffer, a consumer may attempt to take data out of the buffer. If there is no room in the buffer for a producer's data, then the producer will have to wait until space becomes available. If there is not enough data in the buffer to meet a consumer's request, then the consumer will have to wait until the data becomes available. A producer puts as much data into the buffer as possible, then notifies any waiting consumer that more data has arrived and, if it has more data to transfer, puts itself to sleep. A consumer takes out as much data as possible (either the amount it has requested or what is available, whichever is smaller), then notifies any waiting producers and puts itself to sleep if more data is needed. (Note that this solution correctly deals with the case in which a producer has more data to transfer than can be held by an empty buffer). A monitor representing the buffer and the associated produce and consume operations can be defined as:

```
// define buffer monitor.
struct buffer : monitor {
    char *buf ;
    char *in, *out ;
    int nfull, nempty ;
    int alldone ;
    int size ;
```

```

buffer(int);
~buffer()      { delete buf ; }
int consume(char*,int);
void produce(char*,int);
void flush();
};

buffer::buffer(int s) : ("buffer",2)
{
    size = s ;
    buf = new char[size] ;
    nfull = alldone = 0 ;
    in = out = buf ;
    nempty = size ;
}

int
buffer::consume(char* s, int n)
{
    int cnt, cnt2 ;
    int totcnt = 0 ;
    char *lin = s ;
    int bytesleft ;

    enter();

    bytesleft = (int)(size - (out - buf));

    while (n > 0) {
        while (nfull <= 0) {
            if (alldone) {
                exit();
                return(totcnt);
            }
            signalandwait(1, 0);
        }

        cnt = (nfull <= n) ? nfull : n ;
        if (cnt > bytesleft) {
            bcopy(out,lin,bytesleft);
            cnt2 = cnt - bytesleft ;
            bcopy(buf,lin+bytesleft,cnt2);
            out = buf + cnt2 ;
        } else {
            bcopy(out,lin,cnt);
            out += cnt ;
            bytesleft -= cnt ;
        }
        nfull -= cnt ;
        nempty += cnt ;
        lin += cnt ;
        totcnt += cnt ;
        n -= cnt ;
    }
}

```

```

    }
    signal(1);
    return(totcnt);
}

void
buffer::produce(char* s, int n)
{
    int cnt, cnt2 ;
    char* lout = s ;
    int spaceleft ;

    enter();

    spaceleft = (int)(size - (in - buf));

    while (n > 0) {
        if (nempty <= 0)
            signalandwait(0, 1);
        cnt = (nempty <= n) ? nempty : n ;
        if (cnt > spaceleft) {
            bcopy(lout, in, spaceleft);
            cnt2 = cnt - spaceleft ;
            bcopy(lout+spaceleft, buf, cnt2);
            in = buf + cnt2 ;
        } else {
            bcopy(lout, in, cnt);
            in += cnt ;
            spaceleft -= cnt ;
        }
        nempty -= cnt ;
        nfull += cnt ;
        lout += cnt ;
        n -= cnt ;
    }
    signal(0);
}

void
buffer::flush()
{
    enter();
    alldone++ ;
    signal(0);
}

```

With this definition of a bounded buffer, writing the routines which actually do the producing and the consuming is quite straightforward. We need not worry whether or not the routines are monitors since all of the details are abstracted away in the buffer.

```

producer::producer(buffer* buf) : ("producer")
{
    int cnt ;
    char s[80] ;

```

```

    while (1) {
        if ((cnt = read(0,s,80)) <= 0)
            break ;
        buf->produce(s,cnt);
    }
    buf->flush();
}

// define consumer task.
struct consumer : task {
    consumer(buffer*);
};

consumer::consumer(buffer* buf) :("consumer")
{
    int cnt ;
    char s[4];

    while (1) {
        if ((cnt = buf->consume(s,4)) == 0)
            break ;
        write(1,s,cnt);
    }
}

// Main task.
main(int argc, char* argv[], char* envp[])
{
    buffer *b    = new buffer(80);
    producer *p = new producer(b);
    consumer *c = new consumer(b);
}

```

### 2.3. The Queue Classes

Another method of communicating among tasks is to build a one-way pipeline between tasks for the orderly flow of data from one task to another. In the tasking package there is no class named *queue*: instead, there are two classes, *qhead* and *qtail*, which implement the two ends of a queue. This separation allows the producers and consumers of data to be defined without any dependence on each other. A queue can be created with two calls:

```

qhead qh ;
qtail *qt = qh.tail();

```

*qh.tail()* creates a *qtail* and connects it to *qh*. To move data through this queue, use the *qtail* operation, use *put()* to enter data into the queue, and uses the *qhead* operation, *get()*, to retrieve data from the queue. *put()* and *get()* operate on pointers to items of class *object* or items derived from class *object*. Normally, if *get()* operates on an empty queue, the task which issued the request is suspended until the queue is no longer empty. *put()* behaves similarly when the queue is full.

Queues have a maximum size arbitrarily set when the queue is created; the default maximum is 10000. There are also other semantics for handling empty and full queues. By setting a mode value for a *qhead* or *qtail*, the operations can return error messages or values which tell the task that called it of the queue's condition.

Let us look at how to handle the producer/consumer problem by using these queues for communication. Note how the type of data item passed through the queue is defined:

```
// define queue data item
struct item : object {
    int last_message ;
    char s[80];
};

struct producer : task {
    producer(qtail*);
};
producer::producer(qtail* qt) :("producer")
{
    for (item *i = new item ; fgets(i->s,80,stdin) ; i = new item) {
        i->last_message = 0 ;
        qt->put((object*)i);
    }
    i = new item ;
    i->last_message = 1 ;
    i->s[0] = 0 ;
    qt->put((object*)i);
}

struct consumer : task {
    consumer(qhead*);
};
consumer::consumer(qhead* qh) :("consumer")
{
    while (1) {
        item *i = (item*)qh->get();
        if (i->last_message)
            break ;
        printf(i->s);
    }
}

main(int argc, char *argv[], char *envp[])
{
    qhead* qh = new qhead ;
    qtail* qt = qh->tail();

    producer *p = new producer(qt);
    consumer *c = new consumer(qh);
}
```

The Tasking package also supports "splitting" queues in the middle and adding filters to alter the data stream. For example, to change all the lowercase letters to uppercase in the data in the previous example, we can define a task to do this change and add it into the data stream filtering the lowercase letters.

Here is how to split the queue to add the filtering routine. Suppose *qh* and *qt* are already defined:

```
qhead *newqh = qh->cut();
```

```
qtail *newqt = qh->tail();
```

*cut()* splits *qh* and returns a pointer to a new *qhead* which is connected to the tail of the old queue. This can be used as the head for the filter. Now the old head *qh* no longer has a tail associated with it, so it makes a new one, and this tail can be used as the output for the new filter. In this way, the queues have been split and a filter put in the data stream without having to notify the tasks currently using the queue.

Queues can also be spliced back together with the *splice()* operation. To undo what was done above, we can write:

```
newqt->splice(newqt);
```

*splice()* deletes the *newqh*, *newqt* and the queue associated with the tail, and connects their respective heads and tails together to make one queue again.

The producer/consumer problem can be changed as follows to add in the filter to make all lowercase letters uppercase. First define the filter task:

```
#include <ctype.h>
```

```
struct filter : task {
    filter(qtail*,qhead*);
};
```

```
filter::filter(qtail* qt, qhead* qh) :("filter")
{
    while (1) {
        item *i = (item*)qh->get();
        // change all to upper case.
        char c ;
        int t = 0 ;
        while (c = i->s[t]) {
            if (isalpha(c) && islower(c))
                i->s[t] = toupper(c);
            t++ ;
        }
        qt->put((object*)i);
        if (i->last_message)
            break ;
    }
}
```

Next, change the main task to split the queue so that the filter can be added, and add the filter:

```
main(int argc, char *argv[], char *envp[])
{
    qhead* qh = new qhead ;
    qtail* qt = qh->tail();

    // split queue to put new filter in it
    qhead* newhead = qh->cut();
    qtail* newtail = qh->tail();

    filter *t = new filter(newtail,newhead);
    producer *p = new producer(qt);
    consumer *c = new consumer(qh);
```

}  
The queue structure a variety of applications. For example, it can be used as the basis for message-based systems in which a server accepts messages from client tasks and processes them or create other tasks to do so. For further discussion of how the queues work, see [Stroustrup 2].

### 3. The Threads System

The Threads system cheaply supports the concept of a *thread of control* (or *thread*), which is an independent unit of execution, capable of concurrent execution with other threads. Our implementation is on top of workstations running Berkeley UNIX<sup>4</sup> (it currently runs on Suns, MicroVAXes and Apollos) as well as on a shared-memory multiprocessor — the Encore Multimax. It has proved fast enough to satisfy the needs of several projects at Brown and is about to be distributed to researchers at other institutions. The programming interface provided by Threads insulates the user from such details as the number of processors being used: programs written for uniprocessors normally work correctly on multiprocessors. We provide a standard set of high-level programming abstractions and also provide facilities for the programmer to create his or her own abstractions.

The first version of Threads was completed in September 1985, and versions have been used by researchers other than the implementer since then. The first multiprocessor version of Threads was completed in March of 1987, two months after we acquired our Encore. This version has been used by others since April of 1987. A detailed tutorial on the use of the system is available [Doeppner 1]. [Doeppner 2] describes the design of the system.

The notion of cheap concurrency is often known as “lightweight processes.” An early operating system that was based on this notion was Xerox’s Pilot system [Redell]. Other UNIX-based lightweight process implementations have been discussed in the past few years [Binding, Kepecs]. The system that comes closest to ours is Eric Cooper’s C-Threads [Cooper]. Recently an operating-system-supported notion of lightweight process, also known as a thread, has been implemented and used extensively at CMU as part of the Mach system [Tevanian].

What differentiates our system from all of the other UNIX-based systems is its complete support for systems concepts, including I/O, interrupts and exceptions. What differentiates our system from all of the other approaches to inexpensive concurrency is its support for concurrent programming abstractions — both the design of the particular abstractions we have implemented and how we allow the programmer to define new abstractions.

For a highly concurrent style of programming to be practical, threads, which support the concurrency, must be very inexpensive; the overhead required for creating, synchronizing and scheduling threads must be very low. A thread is not a traditional operating-system process, which are typically very expensive to create and very expensive to synchronize. One reason for this expense is that processes are much more than just threads of control. They entail (usually) a separate address space and other protection boundaries which are time-consuming to set up. Another reason for the expense of processes is that they are managed by the operating system kernel; requests to perform operations such as synchronization must be passed to the kernel over a user-kernel boundary that is typically fairly expensive to cross (for example, for Berkeley UNIX running on a MicroVAX<sup>5</sup> II, the overhead for a trivial system call is 200 microseconds).

<sup>4</sup> UNIX is a trademark of AT&T Bell Laboratories.

<sup>5</sup> MicroVAX is a trademark of the Digital Equipment Corporation.

In the Mach system [Tevanian], threads are supported in the kernel, but are cheap enough to qualify as lightweight processes (Mach threads are a lower-level abstraction than our threads). We are very interested in combining our approach with that of Mach, building our threads on top of Mach threads.

Currently all of our Threads system runs as user-mode code. This has resulted in a minimal overhead due to system calls and has allowed our system to be ported fairly easily to other UNIX systems.

Our implementation consists of two layers. The bottom layer, which is built on top of the UNIX process, implements the basic notion of a thread as an independent entity. A thread at this level presents a fairly low-level procedural interface which allows it to be manipulated directly. In the next layer, the thread abstraction is extended to supply the functionality needed to give the programmer the types of programming constructs expected in a high-level language; this layer can be thought of as the runtime library for such a language. A user of the Threads system may add additional layers, building on top of the lower layers by supplying additional procedures and adding additional fields to the thread data structures.

The functionality defined in the bottom layer includes scheduling and context switching, low-level synchronization, interrupt processing, exception handling and stack handling. In addition, this layer provides the routines employed for protection from interrupts and for the locking of data structures when used on a multiprocessor.

In the second layer we build up a set of programming constructs from the low-level interface of the bottom layer. These constructs allow threads to synchronize their execution (using semaphores or monitors [Hoare]), to perform I/O, and to respond to exceptions and interrupts. Exception handling is integrated with synchronization so that, for example, when a thread is forced out of a synchronization construct by an exception, the state of the synchronization construct is "cleaned up" so that it will continue to operate correctly. The programmer may choose a variety of ways of dealing with interrupts. For example, a thread may be created in response to an interrupt or an interrupt may cause an exception to occur in a specified thread.

Recently we have added support for a different type of thread, called a "microthread," which is an extremely low-overhead thread that is useful for such fine-grained applications as parallelizing do loops. They are allocated in groups, so that a set of microthreads will jointly invoke a "parallel function", each of them determining what subtasks they are responsible for and then performing these subtasks. In our Encore implementation, approximately 185 microseconds are required to create a standard thread (vs. 10 — 20 milliseconds to create a process in UNIX) and approximately 5 microseconds are required to start a set of 10 microthreads. We have achieved 90% speedup using microthreads to solve an edit-distance problem. The integration of microthreads into C++ is nearly complete and will be discussed in a subsequent paper.

#### **4. Use of Threads in the Tasking Package**

The concept of parallel tasks in the tasking package maps directly to threads of control in the Threads system. Threads offers all of the constructs needed to create, schedule, synchronize, and maintain tasks. The rest of this section will discuss how threads are used to implement the different parts of the tasking package, including, tasks, queues, monitors, and task wait queues.

##### **4.1. Task Creation and Execution**

A thread consists of a stack and a thread control block which is scheduled on the run queue in the Threads system. For a task to be created, the data structure which controls the task must be associated with the thread's stack and control block. This association is

done at the time the constructor for a class with a base class *task* is called. In the constructor, space for a stack and thread control block is allocated and the calling history on the current task's stack is modified. The new stack is set up so that new thread of control will execute the derived constructor of the task class being created. The current thread's stack is also modified so that it will return directly to the operation which called for the creation of the new task.

All execution in a program using the Tasking Package is done inside the context of some task, even the *main()* routine. The *main()* that is defined by the programmer is actually made into its own task by being called by a library-supplied *main()* routine which also sets up the environment for the Threads system and calls the Threads startup routine *Threadgo*. At startup time, the Threads environment can be changed by the user by passing arguments on the command line which are interpreted in library-supplied *main()*. These arguments primarily affect the members of the class *TASKenv* which defines certain default values and aspects of the Threads environment.

All scheduling issues, including, priorities and preemptive scheduling is handled at the Threads level with no intervention from Tasks. Any changes in the behavior is handled by making Threads calls, such as *THREADsetpriority()* for changing priority and *THREADstopclock()* for turning off preemptive scheduling.

#### 4.2. Task Queues

Task queues are a straightforward use of the Threads queues facility. When a task does a *wait()* on another task it uses Thread's *THREADmovetowaitq()* routine to suspend the thread and put it on a queue. Likewise, the *THREADqueueenext()*, *THREADpullfromq()*, and *THREADmovetorunq()* routines are used to wake up waiting tasks. Also, whenever any of the queues associated with *qhead* or *qtail* need to suspend tasks because of over/underflow conditions, the same Threads routines are used to suspend the tasks.

#### 4.3. The Monitor Class

The *monitor* class is built directly from the monitors of the Threads system: each of the calls in the class map directly to a corresponding Threads call. We would prefer not to require the programmer to supply the *enter* and *exit* calls, since these should be supplied implicitly as part of the abstraction, but we found no convenient method for doing this. However, users of classes derived from the *monitor* class can still view the class as having all of the properties associated with a monitor.

#### 5. Conclusion

The tasking package is a useful set of C++ classes which gives programmers a set of abstractions for creation, maintenance, and synchronization of separate threads of control in a parallel programming environment. The Threads run-time system provides a rich set of constructs for building parallel tasks in a single UNIX process, making implementation of the tasking package an easily manageable job.

#### 6. References

- [Binding] Binding, C., "Cheap Concurrency in C," *SIGPLAN Notices*, Vol. 20, No. 9, September 1985.
- [Cooper] Cooper, E.C., Draves, R.P., "C Threads," Draft of Carnegie Mellon University/Computer Science Report, March 1987.
- [Doeppner 1] Doeppner, T.W. Jr., "A Threads Tutorial," Computer Science Technical Report CS-87-06, Brown University, March 1987.

- [Doeppner 2] Doeppner, T.W. Jr., "Threads — A System for the Support of Concurrent Programming," Submitted for publication, also Computer Science Technical Report CS-87-11, Brown University, June 1987.
- [Hoare] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept", *Communications of the ACM*, Vol. 17, No. 10 (1974).
- [Kepecs] Kepecs, J., "Lightweight Process for UNIX/Implementation and Applications," *USENIX Association Summer Conference Proceedings*, June 1985.
- [Redell] Redell, D.D., Dalal, Y.K., Horsley, T.R., Lauer, H.C., Lynch, W.C., McJones, P.R., Murray, H.G. and Purcell, S.C., "Pilot: An Operating System for a Personal Computer," *Communications of the ACM*, Vol. 23, No. 2, February 1980.
- [Stroustrup 1] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, 1986.
- [Stroustrup 2] Stroustrup, B., "A Set of C++ Classes for Co-routine Style Programming", AT&T Bell Laboratories Computer Science Technical Report, Available with Release notes for 1.2.1 C++.
- [Tevanian] Tevanian, A., Rashid, R.F., Golub, D.B., Black, D.L., Cooper, E. and Young, M.W., "Mach Threads and the UNIX Kernel: The Battle for Control," *USENIX Association Summer Conference Proceedings*, June 1987.



# The Design of a Multiprocessor Operating System<sup>1</sup>

Roy Campbell, Vincent Russo, Gary Johnston

Department of Computer Science  
University of Illinois at Urbana-Champaign  
1304 W. Springfield Ave., Urbana, IL 61801-2987

## ABSTRACT

Evolving applications and hardware are creating new requirements for operating systems. Real-time systems, parallel processing, and new programming paradigms require large adaptive maintenance efforts to modernize existing operating systems. An alternative to such adaptive maintenance is to seek new operating system designs that exploit modern software engineering techniques and methodologies to build appropriately structured modular software. This paper describes an approach to constructing operating systems based on a class hierarchy and object-oriented design, and discusses the benefits and difficulties of realizing this design using C++.

## 1. Introduction

*Choices* is designed as an object-oriented system that supports user applications with an object-oriented operating system interface. The architecture embodies the notion of a *customized* operating system that is tailored to particular hardware configurations and to particular applications. Within one large computing system containing many processors, many different specialized operating systems may be integrated to form a general purpose computing environment. *Choices* is currently implemented on an Encore Multimax.<sup>®</sup>

*Choices*, a *Class Hierarchical Open Interface for Custom Embedded Systems*, provides a foundation upon which to construct sophisticated scientific and experimental software. Unlike more conventional operating systems, *Choices* is intended to exploit very large multiprocessors interconnected by shared memory and/or high-speed networks. Uses include applications where high-performance is essential like data reduction or real-time control. It provides a set of software classes that may be used to build specialized software components for particular applications. *Choices* uses a class hierarchy and inheritance to represent the notion of a family of operating systems and to allow the proper abstraction for deriving and building new instances of a *Choices* system. At the basis of the class hierarchy are multiprocessing and communication objects that unite diverse specialized instances of the operating system in particular computing environments.

The operating system was developed as a result of studying the problems of building adaptive real-time embedded operating systems for the scientific missions of NASA. Major design objectives are to facilitate the construction of specialized computer systems, to allow the study of advanced operating system features, and to support parallelism on shared memory and

<sup>1</sup> This work was supported in part by NASA under grant no. NSG1471 and by AT&T METRONET.

<sup>®</sup> Multimax is a trademark of Encore Computer Corporation.

networked multiprocessor machines. Example specialized computer systems include support for reconfigurable systems, robotics applications, network controllers, aerospace applications, high-performance numerical computations, and parallel language processor servers for IFP[14], Prolog, and Smalltalk. Examples of advanced operating system features include fault-tolerance in asynchronous systems, real-time fault-tolerant features, load balancing and coordination of very large numbers of processes, atomic transactions, and protection. Example hardware architectures include shared memory multiprocessors like the Encore Multimax and networked computers like the Intel Hypercube.

Choices was designed to address the following specific issues: the software architecture for parallel operating systems; the achievement of high-performance and real-time operation; the simplification and improved performance of interprocess communications; the isolation of mechanisms from one another and the separation of mechanisms from policy decisions.

Of particular concern during the development of the system was whether the class hierarchical approach would support the construction of entire operating systems. C++ was chosen because it supported classes while imposing negligible performance overhead at run-time. In particular, we decided to construct all parallel and synchronization features using C++ classes rather than by introducing new language primitives. Thus Choices is also a study of the adequacy of class hierarchies to abstract and support parallelism, synchronization, resource allocation, and other operating system concepts and to allow specializations of classes that facilitate efficient support for applications.

Fortunately for the designers of Choices, there has been a lot of operating system development that is directly applicable to our goals [4]. Abstracting the ideas from many different systems and reorganizing them into an object-oriented system has been a major concern of our design team.

In brief, Choices has been influenced considerably by

- the systems environment provided by UNIX,<sup>o</sup>
- the problem of allowing multiple processors to execute the kernel of an operating system,
- the reduction of process context switching overheads to better support real-time and high-performance applications,
- the open architecture of CEDAR [17],
- the use of classes to extend an efficient systems programming language rather than adopt a more complex programming language that has rich system programming features like Mesa or Ada,
- the exploitation of virtual memory to support object-oriented paradigms and communication primitives [1],
- the avoidance of designing another a distributed operating system but instead concentrating on the design of a parallel operating system,<sup>2</sup>
- the avoidance of the undesirable side effects of cacheing and cache flushing overhead in

---

<sup>o</sup> UNIX is a Registered Trademark of AT&T.

<sup>2</sup> Many distributed/multiprocessor UNIXs (UNIX United [3], LOCUS [11], Mach [1], RFS [13], RIDE [10], NFS [19], Encore Multimax UNIX (UMAX<sup>o</sup>) [8], Sequent Balance<sup>o</sup> 8000 UNIX [15]) still impose UNIX limitations on the parallelism and performance of applications.

cache-oriented multiprocessors,

- the avoidance of the inclusion of specific communication schemes into the lowest structures of an operating system kernel that restrict the possibilities of specializing kernel features to take advantage of communication patterns of the application or communication mechanisms of the hardware,<sup>3</sup>
- the avoidance of overhead from copying messages into and out of virtual memory (some cached systems may pay a double overhead),
- the support of real-time interrupts and global multiprocessor interrupts,
- the inclusion of parallel programming primitives (for example, the parallel creation of parallel processes),
- the provision of appropriate error recovery in parallel processing systems,
- the Clouds [2] notion of a user process accessing a user object,
- the provision of the smallest operating system that will support a particular application on a particular hardware,<sup>4</sup>
- the support for embedded, real-time, and server computing services that are provided by large numbers of fast processors connected together by shared memory and/or by a fast network,
- the support for a computational facility that is multitasked (it supports several concurrent applications), where each task may use multiple processors,
- the support for processes in an application that may have a high degree of communication,

In the subsequent sections, we discuss the class hierarchical organization of Choices, the various classes we have built to implement virtual memory, the concept of process, the notion of a persistent object and exception handling, performance of an object-oriented operating system, and the implementation of the system using C++.

## 2. The Choices Class Hierarchy Model

Several problems emerge when designing an extensible *family* of operating systems where each member can be specialized or customized for a particular application or hardware configuration. Each module within the system may have many different versions tailored for each different member of the family of operating systems. However, since the different versions of a module for different machines or applications all perform a similar function, large portions of different versions of a module will be identical. Customizing an operating system for a new application requires access to particular aspects of the code that may reside in many different

<sup>3</sup> UMAX is a trademark of Encore Computer Corporation.

<sup>4</sup> Balance is a trademark of Sequent Computer Systems.

<sup>3</sup> Message-oriented kernels like the V System kernel [7], Accent [12], Amoeba [18], and MICROS [20] build specific communication schemes into the lowest kernel structures. For example, some systems implement a few ways of providing "virtual" messages like "fetch on access." However, these systems are not easy to adapt to support other approaches like "send process on read" or "remote procedure call on execute."

<sup>4</sup> General purpose operating systems often employ delayed bindings within their architectures to provide flexibility. Examples include communication schemes, file systems and additional kernel code to handle different architectures and configurations. Where several applications need to coexist within the same computing system, Choices allows these applications to each run on their own custom-built Choices operating system. Any communication required between the applications is supported by common Choices primitives and shared persistent objects.

modules.

A class hierarchy provides a solution to these problems. Particular instances of classes in the hierarchy are chosen and combined to produce a customized operating system for a specific architecture and application. Class inheritance provides for code reuse and enforcement of common interfaces. Customization of the operating system for new applications is guided and aided by the structure induced upon the system by the class hierarchy.

A class hierarchy gives more than ease of customization. It also gives us a conceptual view of how portions of an operating system interrelate. It is easier to understand and more flexible than traditional layered approaches to operating system design. A class hierarchy allows conceptual "chunking" of knowledge about portions of a system by learning the function of parent classes and inferring functionality about subclasses. Traditional layered operating system design approaches group large sections of functionality into a layer, but the interrelations of the layers are often complex and poorly understood. Also, changing a piece of a layer is in no way facilitated by the layering itself. However, in a well-designed Class Hierarchical model only the top few classes would need to be mastered to achieve a good overall view of the system. Class derivation gives a method to change specific parts without adversely effecting the whole structure.

Most of the major components of the class hierarchy for Choices are shown in Tables 1 through 4. Table 1 shows the major classes that comprise the first level in the hierarchy. *Object* is specialized into seven subclasses including *MemoryRange* and *SpaceList*. Each subclass redefines and adds new methods<sup>5</sup> to the methods defined for *Object*.

First Level Hierarchy of Classes		
Classes	Methods	
Object	constructor	destructor
↓ MemoryRange	constructor	destructor
↓ Process	constructor	destructor
↓ ProcessContainer	constructor	destructor
↓ Exception	constructor	destructor
↓ Filler	constructor	destructor
↓ SpaceList	constructor	destructor

Key to Hierarchy Tables	
Symbol	Meaning
Method	Definition of Method
↓	Subclass or Inherited Method
Method	Overloading of Method
—	Undefined Method

Table 1: Major Classes at First Level of Hierarchy

<sup>5</sup> In this paper the terms "member function" and "method" are used interchangeably.

Several guidelines have been used to develop an appropriate class hierarchy for Choices. All machine dependencies, operating system mechanisms (for example, page table management), operating system policy decisions (for example, schedulers) and design decisions are encapsulated within objects. In general, a design decision is often represented as a class with subclasses that represent the mechanisms that implement the decision. Table 3 shows an example in which the design separates *ProcessContainers* into containers that hold one thread and containers that hold many. The use of an instance of a subclass of a class as a representative instance of the class is often used to program specific policy decisions. For example, a FIFO scheduling discipline may be imposed on processes being added to a *ProcessContainer* by using an instance of the *FIFOScheduler* class to represent the *ProcessContainer* doing scheduling of processes for the system. A priority scheduled system can be created by replacing this *ProcessContainer* with an instance of a subclass that imposes the priority scheduling discipline on the processes it contains. Wherever possible, the class hierarchy is constructed so that similar sub-hierarchies can be specialized from a common ancestor hierarchy. Thus, for example, there is a hierarchy of classes representing memory management mechanisms shown in Table 2 that is specialized into virtual memory, real memory, and disk storage.<sup>6</sup> Overall, design progressed from class hierarchies that had a large fan out to hierarchies that had a small (2-7) fan out, and greater depth. In the future, multiple inheritance may further refine the hierarchy.

C++ was used to program all parts of the system. The language is efficient and portable. It implements object oriented programming semantics with minimal runtime overhead and thus is ideal for operating system programming. It is also easy to interface C++ to assembler in order to achieve things impossible in the language itself (for example, loading stack pointers and memory management unit registers.) The operating system design requires operating system mechanisms to support classes as objects and the dynamic loading and execution of objects. These facilities are required to build an object-oriented file system, process mechanism and persistent object implementation scheme. The support that C++ provides in writing these systems is limited because dynamic loading is not supported and classes are compiled into C code. However, we do not consider this restriction in C++ a disadvantage in our implementation because, if they had existed, they might have biased our implementation or forced us to modify the semantics of the language.

The Choices classes that support operating system construction are divided into two portions. The *Germ* is a set of classes that encapsulates the major hardware dependencies of Choices and provides an *idealized* hardware architecture to the rest of the classes in the hierarchy. It provides the *mechanisms* for managing and maintaining the physical resources of the computer. A *Kernel* is a collection of classes that supports the execution of applications and implements resource allocation *policies* using the *Germ* mechanisms. Individual customized systems consist of instances of *Germ* classes defined by Choices appropriate for the particular hardware of the system, plus the specifically tailored *Kernel* classes the system builder desires. Once this instance hierarchy is built, individual applications that run on top of the new *Kernel* can further augment the Choices class hierarchy with their own classes.

In Choices, both the *Germ* and *Kernel* class collections are embedded in a class hierarchy that has class *Object* at its root, see Table 1. Class *Object* provides virtual functions for the construction and destruction of system instances.

---

<sup>6</sup> For simplicity, Table 2 does not show the disk storage classes.

In the following sections, we will describe some of the classes that constitute Choices.

### 3. Choices Memory Management

Memory management in Choices is implemented by a class hierarchy derived from the *MemoryRange* class shown in Table 2 and supports virtual memory, the sharing of memory, and memory protection. An instance of a *MemoryRange* represents a contiguous range of memory addresses, as the name implies. *MemoryRange* is subclassed for virtual and real memory. *Space* is a subclass of *MemoryRange* that represents a range of virtual memory that can potentially be addressed by a processor. A *Space* is a range of virtual addresses; however, an address in the range may be unallocated, reserved and invalid, or reserved and valid (where valid implies that it is mapped to physical memory). Another subclass, *Store*, exists to represent physical addresses in a Choices system. The methods operating on *MemoryRanges* are specialized for *Spaces* and *Stores* and some of them are also shown in Table 2. Many of the methods defined for *Spaces* are inherited by the subclasses of *Space*. The methods are summarized below.

#### 3.1. MemoryRange Methods

The constructor for a *MemoryRange* takes, as a minimum, a base address and length for the range. The parameters of the constructor are augmented in the various subclasses of *MemoryRange* in order to implement the various specializations of this class.

The most important methods for *MemoryRanges* are **reserve** and **release**. Reserve records that a sub-range of addresses within the *MemoryRange* are in use or "reserved". The sub-range to reserve is specified by a starting address and a length argument. Reserve returns an error if any of the addresses are already reserved (and have not yet been released). A method **allocate** (not shown) operates like reserve but takes a single argument, the length to reserve, and reserves a range of unused addresses starting at an address selected by the method rather than by an argument. Functions **isReserved** and **isAvailable** exist to test whether a sub-range is already

Hierarchy of Memory Classes									
Classes	Methods								
<i>MemoryRange</i>	<b>resrv.</b>	<b>release</b>	<b>phys.Ad</b>	<b>isResrv.</b>	<b>isAvl.</b>	<b>start</b>	<b>end</b>	<b>size</b>	—
↓ <i>Spaces</i>	<i>resrv.</i>	<i>release</i>	<i>phys.Ad</i>	<i>isResrv.</i>	<i>isAvl.</i>	<i>start</i>	<i>end</i>	<i>size</i>	<b>FixF't</b>
↓ ↓ <i>F'tInSpace</i>	<i>resrv.</i>	<i>release</i>	<i>phys.Ad</i>	<i>isResrv.</i>	↓	↓	↓	↓	↓
↓ ↓ ↓ <i>FilledF'tInSpace</i>	<i>resrv.</i>	↓	↓	↓	↓	↓	↓	↓	<i>FixF't</i>
↓ ↓ <i>PrefetchedSpace</i>	<i>resrv.</i>	↓	↓	↓	↓	↓	↓	↓	↓
↓ <i>Stores</i>	<i>resrv.</i>	<i>release</i>	<i>phys.Ad</i>	<i>isResrv.</i>	<i>isAvl.</i>	<i>start</i>	<i>end</i>	<i>size</i>	—

Hierarchy of SpaceList Classes					
Classes	Methods				
<i>SpaceList</i>	<b>add</b>	<b>remove</b>	<b>isIn</b>	<b>spaceContainer</b>	<b>setEqual</b>
↓ <i>Domain</i>	<i>add</i>	<i>remove</i>	↓	↓	↓
↓ <i>Universe</i>	<i>add</i>	<i>remove</i>	↓	↓	<i>setEqual</i>

Table 2: Memory and SpaceList Class Hierarchies

reserved or not.

Other `MemoryRange` methods include `start`, `end` and `size` that return the base address of, the last address of, and the total number of addresses in the range. A function `isIn` (not shown) returns whether an address lies within the `MemoryRange`.

### 3.2. Stores

The `Store` class represents a range of *physical* or *real* memory. It encapsulates the data structures used by the `reserve`, `allocate`, and `release` methods of `Store` to manage the allocation of real memory to a process. A Choices operating system may include multiple instances of the `Store` class; each representing a physical memory with a different property or attribute. For example, a `Store` instance may represent the local memory private to a processor, a software maintained private cache, the physical memory shared within a multiprocessor, or the physical memory shared between multiprocessor clusters.

### 3.3. Spaces

The `Space` class represents a range of *virtual memory* that may be shared and protected. An instance of `Space` is similar to the notion of a segment, it may, for example, represent a stack, data, or code segment. A `Space` encapsulates the hardware dependent data structures (for example, the page or segment table entries) that implement the mapping of the virtual addresses in the range to physical memory. The methods `reserve`, `allocate`, and `release` are overloaded to manage the allocation of virtual memory addresses. For example, a request to extend a run-time stack will result in the reservation of the virtual memory addresses for that extension. `Spaces` augment the set of methods inherited or redefined from `MemoryRange` with `fixFault`. The `fixFault` method is used to implement demand paging, prepaging, or segmentation. For example, the Encore Multimax version of `fixFault` provides demand paging and is invoked by the exception handling mechanism of a processor whenever a memory referencing error, including a page fault or protection violation, is detected by the hardware. The `fixFault` method validates or maps real memory into the virtual memory addresses in page-size increments. Should the virtual memory addresses correspond to virtual memory that has been paged-out, the method initiates reading the pages from disk into the appropriate real memory. A `Space` also maintains the access permissions for its virtual address range. Methods are provided to set these permissions. The permissions on a `Space` apply to all the addresses represented by that `Space`.

Every `Space` maps its virtual addresses by referring to a lower-level `MemoryRange` that is its source of real memory. This `MemoryRange` may be a `Store` or, recursively, another `Space`. A "low-level" `Space` maps virtual addresses into the real memory of a `Store` using the methods of the `Store` to acquire valid real memory addresses. A "high-level" `Space` maps virtual addresses into the real memory of a `Store` through an intermediary `Space`. Recursively, it uses the `MemoryRange` methods on the `Space` to acquire valid real memory addresses. The `MemoryRange`, `Space`, and `Store` method `physicalAddress` returns a real memory address corresponding to a virtual address argument. The `physicalAddress` method for a `Space` uses the internal virtual to real memory address mappings of the `Space` to translate valid virtual addresses into corresponding physical addresses. If the virtual address is unmapped (invalid), the `Space`'s method invokes the `physicalAddress` method of the next lower-level `MemoryRange` and returns the result of that method. An invocation of a `physicalAddress` method on a `Space` can result in a chain of such invocations. This chain continues until either a `Space` is reached for

which a valid mapping for the virtual address is defined or a Store is reached<sup>7</sup>. The `physicalAddress` method for a Store both reserves real memory and returns its physical address in a similar manner to the `allocate` method. Thus, eventually, all `physicalAddress` requests are satisfied.

Spaces allow several processes to share memory. If required, each process may have a different access permission to that memory. The simplest mechanism for sharing a memory is for each process to share a common Space. However, using this approach each process will have the same access permissions to the memory<sup>8</sup>. To overcome this limitation, processes may share memory by having multiple Spaces, each of which implements a different access permission. For example, two processes may each have a Space that maps a given virtual address range into the same real memory locations. Each Space may set up its mapping tables with different access permissions. One process may be granted read/write access to the memory and another may be granted read only access. If a page of memory has to be paged out, both Spaces must invalidate the corresponding virtual memory address range. Spaces also allow a real memory to occupy different virtual address ranges. For example, a given set of memory locations can be shared by mapping it into both a virtual address range of  $0 - n$ , and a virtual address range of  $m - m+n$ .

### 3.4. Space Lists, Domains and Universes

A *SpaceList* provides methods for the aggregation of Spaces and is shown in Table 2. It is specialized into a *Domain* that represents the virtual memory that can be accessed by a user process as it executes an application or a persistent object method. Domains have methods `add` and `remove` that grant or revoke a process' access to particular data and code. In addition, methods are provided to check if a Space is contained within a Domain (`isIn`), and to return the Space within a Domain containing a given address (`spaceContaining`). Another specialization of a *SpaceList* is the *Universe* that represents the virtual memory and, in particular, the actual hardware translation mechanism of a processor. A Universe is a list of the non-overlapping Spaces that form the virtual memory that is addressable by a processor at any one time. Domains as well as individual Spaces may be added or removed from a Universe. In the Multimax implementation, the Universe maintains the first level page tables and is responsible for keeping the actual memory mapping of the processor consistent with its list of Spaces.

#### 3.4.1. Primitive and Derived Spaces

A process may have rights to access a Space as a *Primitive Space* that contains memory (for example, a process stack, code, or local data), or as a *Derived Space* that contains persistent objects<sup>9</sup>. Primitive Spaces are protected from invalid read, write, or execute access. The contents of a Derived Space cannot be accessed by a process unless it changes its domain of execution to a persistent object that is encapsulated within the Space. This change of domain can only occur by the invocation of a persistent object method. Such an invocation creates a page fault. The Germ then checks that the method invoked is a valid method and that access to that method has been granted to the process. A Domain containing the Primitive Spaces that permit access to the

<sup>7</sup>Usually, there is one Space that is at the lowest-level and this coordinates the reservation and validation of the shared memory.

<sup>8</sup>It is possible to overcome this problem by running some processes in supervisor state and others in user state. This solution is used for certain system functions, but is not a general mechanism.

<sup>9</sup>A Derived Space is created from a Primitive Space by granting processes access rights to the methods of the objects within the Space. In Choices, such objects are called "persistent" because their existence becomes independent of the lifetime of any one process (in particular the one that created it). We emphasize the distinction between a Derived Space and a persistent object. Although a Derived Space can contain persistent objects, the Space itself is a Germ object.

contents of the persistent object is then added to the Universe and the method is executed. The change in Domain may also remove other Domains from the Universe according to protection policies. (However, a discussion of the implementation of protection policies and the naming schemes for persistent objects and their methods is beyond the scope of this paper.)

The next section discusses the Choices concept of a process.

#### 4. The Choices Process Model

Choices was designed to support real-time multiprocessing and parallel computing on large numbers of processors. To facilitate this, Choices supports the concept of a computation that is composed of a potentially large number of lightweight, independent parallel processes. A single one of these processes is represented by an instance of the Choices *Process* class shown in Table 3. An application may use multiple communicating processes to achieve concurrency and parallelism. Each Process represents a small independent sequential computation that can share memory through the memory management mechanisms previously described. Processes exist orthogonally to memory and address management issues. Each Choices Process has a Domain that specifies its associated virtual memory. Usually, the executable code, initialized data, uninitialized data, and stack are represented as separate Spaces within this Domain. The constructor for a process is parameterized by an initial Domain, an initial program counter and stack pointer, and arguments to the process. Methods for Processes alter their Domains, manipulate scheduling parameters, and handle preemption and dispatching.

##### 4.1. Process Context Switching

The state of a process is recorded by storing a stack pointer, a program counter, and a set of register contents within an Process object. A small system stack is maintained by a Process in order to handle hardware preemption. The **dispatch** method reloads a CPU's registers with copies that are stored within the Process. In the Multimax implementation, if the Domain of the Process being dispatched matches the Universe of the processor (the processor may have been executing a process that has a similar domain), no memory context switching is necessary. That is, the Multimax page table entries do not need to be modified and the memory management unit (MMU) does not need to be flushed.

Interrupt and real-time processing require the ability to switch between processes with minimum context switching overhead. Unfortunately an executing process accesses a stack, code, and data represented by the various Spaces contained within its Domain. To accommodate high-performance context switching, processes may lock the memory of a MemoryRange as resident (the locking methods are not discussed in this paper.) Locking memory to be resident within a Space causes the corresponding virtual addresses to be validated and the associated real memory to be locked as resident in physical memory by the corresponding Store. In addition, a process may lock a Space within the CPU's Universe, making the Space and its tables resident and ensuring that the Space's virtual memory mapping is available. A context switch to a process that addresses only resident pages in resident virtual memory creates only the register loading overhead.

Interrupt handlers and real-time processes can be implemented using this high-performance optimization, if desired. Such processes may still be protected from other applications by running the processes in the privileged state of the processor and setting the memory protection of the Spaces in their Domain to exclude access in non-privileged mode. Thus, even though a Space may be locked in the virtual memory of the processor, it can remain protected from unprivileged processes. The Kernel memory of a Choices system is implemented as one such Space.

## 4.2. Interprocess Communication

Communication between processes can be achieved by means of shared Spaces or by invoking methods<sup>10</sup> on another process's Process. Popular shared memory and message passing communication schemes exist in the system as part of the operating system and are defined within the class hierarchy (not shown in this paper.) Other user defined communication schemes can be built by extending the class hierarchy. An interface compiler for C++ enriches the possible communication schemes. Currently defined are a Path C++ class (named after Path Pascal [5]), and semaphores. Monitors, messages, and simple varieties of guarded commands are currently being designed and implemented.

Protected communication can be achieved by means of shared Spaces containing persistent objects (defined later). The methods of such objects may enforce particular communication protocols upon the processes that use them and the protection provided by the objects methods prevents misuse.

Persistent objects may be *active*, that is, they may have constructors or methods that create "encapsulated" Processes. Such a Process is initialized with a Domain that includes the primitive Spaces of the persistent object. Thus, a process can be directly associated with a persistent object. Active objects can be used to implement name servers and to send asynchronous messages. Several persistent *system* objects augment the shared persistent objects and provide high-performance communication channels between processes and between processes and devices. System objects can support stream-based communications, broadcasts, multicasts, and block I/O.

## 4.3. ProcessContainers and Scheduling

Scheduling and dispatching issues in Choices are handled by instances of the *ProcessContainer* class shown in Table 3. A ProcessContainer, as the name implies, is a container of

Hierarchy of ProcessContainer Classes					
Classes	Methods				
ProcessContainer	add	remove	isEmpty	—	—
↓ SingleProcessHolder	add	remove	isEmpty	—	—
↓ ↓ LockedSn'glPrc'sHl'dr	add	remove	↓	—	—
↓ ↓ CPU	add	remove	↓	disableIntrpt.	enableIntrpt.
↓ FIFOScheduler	add	remove	isEmpty	—	—
↓ ↓ RRScheduler	↓	remove	↓	—	—

Hierarchy of Process Class					
Classes	Methods				
Process	dispatch	domain	ch'gDomain	sch'Info	setSch'Info

Table 3: Process and ProcessContainer Class Hierarchy

<sup>10</sup> Such a method is similar in intent to the UNIX signal.

Processes. It is the basic entity of scheduling and dispatching in any Choices system. All scheduling issues involve moving Processes between ProcessContainers.

The queueing discipline that a ProcessContainer uses depends entirely on the subclass of the ProcessContainer hierarchy being used. The top level ProcessContainer class itself is abstract and only defines the operations **add** (for inserting Processes into the container), **remove** (for removing the next available Process from the container), and **isEmpty** (for testing whether the container is empty or not.) Subclasses redefine these methods, for example, to add and remove Processes in FIFO, LIFO, or priority order (not shown.)

ProcessContainers implement the "traditional" run/ready/blocked queue models of operating systems. ProcessContainers are also used to store Processes that await an event or are blocked on a semaphore operation. A special subclass of ProcessContainer, *CPU*, has an **add** method that "stores" and executes a Process on a processor. The CPU **remove** method is used to model preemption. The CPU is conceptually a special type of ProcessContainer that invokes the **dispatch** method on a Process that is placed into it. Multiprocessing fits naturally into the Choices model of scheduling since a Choices system can consist of more than one CPU object. Persistent objects are discussed further in the next section.

## 5. Persistent Objects

Choices is designed with the objective of placing many operating system and subsystem components in a protected Space rather than in a kernel as is done in traditional systems. This is done to reduce the interdependences among operating system components and to increase the coherence of the components themselves. Such components are implemented as Choices *persistent objects*. That is, instances of classes that reside in memory for periods that exceed the execution of a particular process and that may be shared between multiple processes. Persistent objects may be mapped into the virtual memory of several processors at the same time using the Space shared memory implementation. In a sense, the Germ and Kernel of a Choices system are collections of persistent objects that are always resident and accessible (in a controlled manner) in the address space of every processor.

A full description of the protection scheme used in Choices is beyond the scope of this paper. However, we must introduce enough of the scheme here in order to describe access to (and the invocation of methods on) a persistent object. Each process executes within a protection domain that dictates what the process may access. The protection domain of a process is dynamic and may change by adding or removing Spaces. Initially, the protection domain depends upon the protection of the executable file that the process is created from and the protection domain of the parent process. A process that executes a method of a persistent object enters a new protection domain that depends upon the protection of the Derived Spaces containing the object and the protection domain of the process. When the process returns from the method invocation, its previous protection domain is restored. The scheme is implemented using the memory management classes introduced in §3.

For example, policy modules of the operating system that traditionally are part of the kernel, may be implemented as persistent objects. A process executing one of the methods within these persistent objects may require access to Germ data structures (typically also requiring some sort of "supervisor" execution privileges). This is possible by having the process enter the protection domain of the persistent object (which would include the change to the supervisor execution level) via the *gate* mechanism. The gate mechanism implements the controlled entry of the process into the new protection domain. When the invoked persistent object method returns, the process' protection domain is restored to what it was before the method call.

Processes access persistent objects using an *object descriptor* and a method. A process must obtain the object descriptor before use. Object descriptors are provided from user or system name servers.

Name servers are themselves persistent objects. Choices includes "standard name servers" that are in the Kernel and may be accessed by every process. These name servers provide basic facilities like the standard file system and intertask communication. Other user defined name servers must be accessed through the standard name server utilities.

On request, the name server grants the process access to the object and returns an object descriptor for the requested object. The grant operation is implemented in the Germ and checks Kernel protection policy to determine if the name server/process grant operation is valid. The name server must have appropriate access rights to the persistent object. If the operation is valid, the Germ adds the Space of the persistent object to the Domain of the Process, and returns the Space address and gate information to the name server. The name server packages an object descriptor which includes the persistent object, Space and gate information and returns.

An operation on a persistent object is invoked through a *gated request*. The Germ ensures that the object descriptor and method used by the process' gated request correspond to the valid persistent object address and method entry point within the Space. The Domain of the Process is changed to reflect the protection domain requirements of the Space.

In hardware architectures with limited virtual memory, the gated method of invoking a persistent object allows many different Spaces to share the same virtual memory address range. The Space and the persistent objects it contains can be mapped into and out of the same address range on demand<sup>11</sup>. In such implementations, the Domain will contain each Space, but only one of the Spaces will be present in the Universe at any one time.

## 6. Exception Handling in Choices

Exceptions in Choices are managed by the *Exception* class and its various subclasses shown in Table 4. The parent class of Exception defines the method, **raise**, to manage or correct the exception condition. Upon an exception condition, the Choices Germ manages the task of converting the machine dependent details of exception processing into an invocation of the raise method for the Exception object managing the exception.

Two subclasses of Exception of interest are *Trap* and *Interrupt*. The Trap class provides Choices with a mechanism for handling traps that a process may generate as a direct result of its execution. This includes machine traps (for example, divide-by-zero and illegal instruction), virtual memory access and protection errors (for example, page faults of various types), and explicit program traps (for example, a "system call".)

The basic function of a Trap handler is, if possible, to service the exception condition within the context of the faulting Process, or otherwise to terminate the execution of the faulting Process.

Interrupts occur asynchronously and, in general, have nothing to do with the currently executing process. In Choices, the await method of an Interrupt can be used to specify a Process that must be executed when the Interrupt is raised. (Interrupts *must* be awaited if they are not to be missed). The raise method of the Interrupt class saves the context of the interrupted process and

<sup>11</sup> In many hardware architectures, a persistent object must be relocated by a link editor to allow it to execute within a specific address range. This implies that once it is activated, it cannot be moved to a new address range.

Hierarchy of Exception Classes		
Classes	Methods	
Exception	<b>raise</b>	—
↓ HardwareException	↓	—
↓ ↓ AbortTrap	<b>raise</b>	<b>fixdPgFlt.</b>
↓ ↓ SVCTrap	<b>raise</b>	—
↓ ↓ IllegalInstructionTrap	<b>raise</b>	—
↓ ↓ UndefinedInstructionTrap	<b>raise</b>	—
↓ ↓ DivideByZeroTrap	<b>raise</b>	—
↓ ↓ InterruptException	<b>raise</b>	<b>await</b>
↓ ↓ TimeSliceInterrupt	<b>raise</b>	<b>clockTick</b>
↓ SoftwareException	<b>raise</b>	<b>handler</b>
↓ ↓ TwoProcessContainerException	↓	<i>handler</i>
↓ ↓ OneProcessContainerException	↓	<i>handler</i>
↓ ↓ GarbageException	↓	—

Table 4: Exception Class Hierarchy

resumes the Process awaiting the occurrence of the interrupt. The Choices Germ has no requirement that all interrupts be handled by the class Interrupt. A Choices kernel implementer can choose to have any type of Exception object handle an interrupt. In future work, various user-oriented exception schemes will be implemented as classes and by the interface compiler. Examples of such schemes can be found in [6].

## 7. Experiences Using C++ as a Systems Programming Language

It is not often that it is possible to conduct operating systems research using a new language. Since the C++ compiler<sup>12</sup> generates C and we were consequently able to port it to the Encore Multimax in just a few hours, we were able to do all our development in C++. No additional vendor support for C++ was required. In addition, we were able to exploit the Encore C compiler for the Multimax which produces highly optimized code. Because we are using the same C compiler for measuring performance of algorithms on the Multimax under UNIX and under Choices, our measurements more accurately reflect the differences in the operating systems rather than in their compilers.

Overall, the implementation of Choices has benefited greatly from the availability of C++, its performance, and its compiler. Run-time overhead is negligible, even in the case of *virtual* member functions (class methods). Because of the small overhead, the authors feel that virtual member functions could have been defined as the default in the language since their semantics more closely model the desired behavior of method (member function) redefinition in class hierarchies.

We have encountered some minor problems and issues that have not been a major cause for concern but, from our (perhaps naive) perspective, could be addressed and improved in future

<sup>12</sup> The AT&T C++ Translator.

releases of C++.

The order in which the constructors for *statically allocated* objects are invoked should be under the control of the programmer (or at least defined). For example, during the boot of Choices we have preferred to initialize the Console object before other objects in order to permit the constructors of other system objects to print diagnostic messages.

We believe that classes should be “first-class” objects. That is, a class should be an entity on which a method can be invoked. This would provide a number of advantages. The existence of *class* data and methods would be more consistent with the way in which *instance* data and methods are modeled.<sup>13</sup> For example, **new** and **delete** could be class methods. This would allow easier (and cleaner) customized memory allocation strategies to be implemented on a per-class basis.

Next, class objects would simplify the implementation of dynamic method binding. For example, virtual function tables (which currently do not have unique instances) could be stored as class data. Each instance of a class could include a reference to its class object, so that a change to the class object’s virtual function table could be used to change the binding of a method of every instance of the class.

Last, class objects would allow a solution to the static constructor ordering problem. Constructor dependency information could be included in the class data, and could then be topologically sorted at link-time to determine a correct calling order.

## 8. Summary

A Choices Kernel currently runs on a 10 processor Encore Multimax that supports the MemoryRange model of memory management as well as the Process and Exception concepts. like data reduction or real-time control.

Of particular concern during the development of the system is whether or not the class hierarchical approach can support the construction of *entire* operating systems. C++ was chosen as an implementation language because it supports class hierarchies and inheritance while

Preliminary Choices Performance Data		
Encore Multimax 32032 (0.75 MIP)		
Operation	Encore 4.2 BSD Unix	Choices
System Call Overhead	173μsec	39μsec
Process Creation	26.3msecs	3.8msecs
Context Switch	—	536μsecs
Shared Memory Example <sup>14</sup>	0.032secs	0.022secs

Table 5: Performance Data

<sup>13</sup> Currently, class *data* are obtained using the keyword “static” in the class definition (there are no class *methods*.)

<sup>14</sup> The example creates four processes on independent processors, three sum a ten column array and the fourth sums the three resulting sums. The Multimax multitasking library package was used under UMAX.

imposing negligible performance overhead at run-time. A software monitor is being used to evaluate the performance of Choices on an Encore Multimax with DPC processors. Although it is difficult to provide a meaningful performance measurement of an operating system, we have obtained results that are encouraging and these are shown in Table 5. System call overhead (including a trap and change to supervisor state) compares favorably with UNIX and is only about four times the overhead of a normal procedure call. The process creation time includes creation of new virtual memory "spaces" for the process. Further tuning will improve these figures.

Current effort is devoted towards improvement and further implementation of communication and persistent object support. Future plans include an object-oriented file system, an advanced interface compiler, and tools for configuring Choices systems. Once Choices is stable, the code will be placed in the public domain to promote research into customized operating systems.

## References

1. Accetta, Mike, et. al. "Mach: A New Kernel Foundation for UNIX Development." USENIX Conference Proceedings, June 1986, pages 93-111.
2. Allchin, J. E. and M. S. McKendry. "Support for Actions and Objects in Clouds: Status Report." Georgia Institute of Technology Technical Report GIT-ICS-83/1, Atlanta, Georgia, January 1983.
3. Brownbridge, D. R., L. F. Marshall, and B. Randell. "The Newcastle Connection, or UNIXes of the World Unite!" Software - Practice and Experience, 1982, pages 1147-1162.
4. Campbell, Roy H., Gary M. Johnston, and Vincent F. Russo. "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)." Operating Systems Review, Vol. 21, No. 3, July 1987, pages 9-17.
5. Campbell, Roy H. and T. J. Miller. "A Path Pascal Language." Department of Computer Science Technical Report UIUCDCS-R-78-919, University of Illinois at Urbana-Champaign, Urbana, Illinois, April 1978.
6. Campbell Roy H. and B. Randell, "Error Recovery in Asynchronous Systems." IEEE Transactions on Software Engineering, Vol. SE-12, No. 8, August 1986, pages 811-826.
7. Cheriton, David R. and Willy Zwaenepoel. "Distributed Process Groups in the V Kernel." ACM Transactions on Computer Systems, May 1985, pages 77-107.
8. Encore. "Encore Multimax Technical Summary." Encore Computing Corporation, 1986.
9. Li, Kai and Paul Hudak. "Memory Coherence in Shared Virtual Memory Systems." Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing, August 1986, pages 229-239.
10. Lu, P. M. "A System for Resources Sharing in a Distributed Environment--RIDE." Proceedings of the IEEE Computer Society Third COMPSAC, IEEE Press, New York, 1979.
11. Popek, G., B. Walker, et. al. "LOCUS: A Network Transparent High Reliability Distributed System." Proceedings of the Eighth Symposium on Operating Systems Principles, printed as Operating Systems Review, Vol. 15, No. 5, December 1981.
12. Rashid, Richard F. and George G. Robertson. "Accent: A Communication Oriented Network Operating System Kernel." Proceedings of the Eighth Symposium on Operating Systems Principles, printed as Operating Systems Review, Vol. 15, No. 5, December 1981, pages 64-75.
13. Rifkin, Andrew P., et. al. "RFS Architectural Overview." USENIX Conference Proceedings, Atlanta, Georgia, 1986.
14. Robison, Arch. D. "A Functional Programming Interpreter." M.S. Thesis and Department of Computer Science Technical Report UIUCDCS-R-87-1714, University of Illinois at Urbana-Champaign, Urbana, Illinois, March 1987.
15. Sequent. "Balance 8000 Guide to Parallel Programming." Sequent Computer Systems, July 1985.
16. Snyder, Lawrence. "Formal Models of Capability-Based Protection Systems." IEEE Transactions on Computers, Vol. C-30, No. 3, March 1981.

17. Swinehart, Daniel, Polle Zellweger, Richard Beach, and Robert Hagmann. "A Structural View of the CEDAR Programming Environment." *Transactions on Programming Languages and Systems*, October 1986, Vol. 8, No. 4, pages 419-490.
18. Tanenbaum, Andrew S. and Sape J. Mullender. "An Overview of the Amoeba Distributed Operating System." *Operating Systems Review*, July 1981, pages 51-64.
19. Walsh, Dan, et. al. "Overview of the Sun Network File System." *USENIX Conference Proceedings*, January 1985, pages 117-124.
20. Wittie, L. D. and A. Van Tilborg. "MICROS - A Distributed Operating System for MICRONET - A Reconfigurable Network Computer" in *Tutorial: Microcomputer Networks*, H. A. Freeman and K. J. Thurber, eds., IEEE Press, 1981, pages 138-147.
21. Wulf, William A., et. al. "HYDRA: The Kernel of a Multiprocessor Operating System." *Communications of the ACM*, June 1974, pages 337-345.



# C\*: A C++-like Language for Data-Parallel Computation

John R. Rose  
Thinking Machines Corporation  
245 First St.  
Cambridge, MA 02142

(617) 876-1111  
rose@think.com, ihnp4!think!rose

December 7, 1987

## 1. Introduction

C\* is a language that extends the C language, in ways similar to the C++ extensions, in order to support *data parallel* programming. In the data parallel programming style, the programmer writes code which acts on a single data element. This code is then run in parallel on a large set of similar data elements. During execution of the program, there is always a current *active set* of data elements to which the parallel actions apply. This active set may be expanded, reduced, or replaced under program control. Parallel actions are highly synchronized, and at any time there is only one active thread of control.

For example, a data element might be a database record, and the code to process it might perform a pattern match on a record. The active set could be those records which match the query so far; as successive subparts of the query are executed, the active set would grow smaller. Generally, data parallel programming is suited to applications which have a modest amount of code and large amounts of data. Therefore, it maps well to massively parallel architectures like the Connection Machine system.

In this note, we will examine the abstract machine which executes a C\* program, briefly enumerate the principal language extensions C\* makes over C, describe the type system of C\*, and its relation to that of C++. After that we will list the main differences between C\* and C++, and finally there will be some example code.

## 2. Machine Model

C\*, like any programming language, has an *abstract machine* on which the language is envisioned to run. To start with, the C\* machine includes a C machine. This sub-machine is called the *front end*, and is responsible for manipulating scalar quantities. There is also an unlimited number of smaller *parallel processors* which handle parallel values. Each of these is also a little C machine, with its own local address space, separate from the storage of any other processor in the system. We call this storage *parallel-processor-local*, or just *processor-local*.

Pointers can be formed to any data element in storage, but the machine distinguishes a pointer into the front end (a regular C pointer) from a pointer to a particular datum on a particular parallel processor. The former kind of pointer is called *mono*, and the latter is called *poly*. This distinction shows up in the typing rules of the C\* language, as we will see later. We must also distinguish processor pointers, and pointers which locate data in processor-local storage without naming a specific processor. Poly pointers consist of ordered pairs of processor pointers and processor-local pointers. Finally, there is also a *system* pointer (less well supported by hardware) which can range over both the mono and poly address spaces. These pointers are used to fetch and store data in the ordinary way. There are no restrictions on the flow of data, from point to point, anywhere within the system.

The data parallel model calls for synchronous, single-threaded execution of the program. This means that processors in the active set must have similar memory layout. For example, if a negation operator is executed, it will apply to a single processor-local memory address, at all processors in the active set. Processors which use the same processor-local addresses in different ways cannot be in the same active set. This feature of the machine will be reflected in distinction between different processor types.

---

Connection Machine is a registered trademark, and C\* is a trademark, of Thinking Machines Corporation.

### 3. Language Extensions

The name "C\*" is intended to look a bit like a regular expression, reflecting the fact that there are any number of C processors available. Many of the language's features are devoted to organizing the separate C programs which run on these processors.

Because each parallel processor is a little C machine, it has storage for many data values. Therefore, C\* uses aggregate types, called *domains*, to describe processors. Each processor is represented in C\* by an *instance* of a particular domain. The keyword "domain" works syntactically like "struct", "union", or "class", and is used to group together any number of data values into a domain. Multiple domains are supported, and there is single inheritance as with C++ classes. Every domain instance is implemented by a parallel processor, in whose processor-local memory are stored values for the member variables of that domain. The typing rules associate a domain with each active set, and only the members of that domain are in scope for the code operating on that active set. This enforces the consistent use of processor-local storage required by the machine model.

Since each processor in the C\* model is a full-fledged C machine, code written inside a domain declaration can contain any form valid at top level. For example, here is some code which might be used to investigate the theory of certain groups, or perhaps play blackjack:

```
const NCARDS = 52;
domain Card {
    typedef domain Card* Permutation;
    Permutation next_card, prev_card;
    int suit, value;
    extern void shuffle(Permutation);
} deck[NCARDS];
```

Just as top-level forms define computations for the front end, forms within a domain declaration define computations which are carried out by active sets of parallel processors. In particular, the "extern" and "static" keywords may be used to create data, and type definitions and functions are allowed within a domain declaration. Finally, and most unlike structs, unions, and classes, a domain declaration need not exhaustively specify the members of the domain. Domain declarations accumulate, within files and across files. This feature is required by the potential size and complexity of a parallel processor.<sup>1</sup> Here is a declaration which could accumulate with the one above, perhaps in a different file:

```
domain Card {
    typedef domain Card* Permutation;
    extern suit, value;
    static mark;
    void shuffle(Permutation x) {
        shuffle_1(x); mark_cards();
        ace_to_bottom_of_deck();
    }
};
```

C\* has the usual rules about promoting scalar values to parallel: If one operand of a binary operation is parallel, the other operand is replicated (that is, loaded into processor-local memory) as necessary.

As in C, pointers can be formed to all data objects (except fields and registers), and any processor can hold a pointer anywhere. Therefore, most inter-processor communication patterns are just specific cases of loading and storing through C pointers. There is an important class of exceptions: Collisions are possible. Informally, this happens when there aren't as many destinations as sources. For example, storing a parallel value into a mono register requires (in general) some method for choosing *which* single value is to be stored. Such methods are called *reductions*, because they reduce many values to a single value. C\* allows all the usual commutative operators to be used to perform reductions. The

---

<sup>1</sup> Recall that poly addresses are ordered pairs of processor address and processor-local memory address; this means that processor sizes do not play an important role at compile time.

notation for this uses the C compound assignment operators reinterpreted by the *As If Serial Rule*. Under this rule, source values are assigned to the destination one at a time, as if in some serial order. For example, in "Y += X", if there are many values of "X" and one destination "Y", all "X" values are summed together into "Y", along with the original "Y" value. In more detail, a compound assignment is interpreted using the following steps. First both operands are replicated if necessary; the source would be replicated as an lvalue. There are collisions if and only if the source gets replicated. Next, the compound assignments are done for each source/destination pair, as if in some serial order. The exact order is defined by the implementation, which is also free to use an efficient algorithm, such as parallel evaluation of a balanced tree of partial results. This rule extends the usual C rule, and produces useful behavior in the event of collisions. There are also unary versions of the compound assignment operators; for these the compiler supplies an appropriately-initialized destination. Thus, "(|=X)" computes the bitwise inclusive OR of all "X" values in the active set.

Minimum and maximum operations are missing from standard C. They are very useful in writing serial code; witness their frequent definitions (as macros) in C source files. But min and max are of greater importance as reduction methods, because they provide a deterministic choice of one of a set of values. Therefore, C\* adds min and max operators (spelled "<?" and ">?") with the corresponding compound assignments. There is also a compound assignment for the comma operator (",=") which means "in case of collisions, choose an arbitrary representative".

C\* has a very powerful model for control flow. Control flow becomes parallel when a parallel value is encountered in the position of a loop or conditional predicate, or a switch selector. From the *local* viewpoint of a single processor, control flows where it should; if that processor got a false predicate value, the loop terminates, or the conditional fails locally. From a *global* viewpoint, the compiler manages the serial execution of multiple threads, each being run by a subset of the active set. For example, when a parallel if is encountered, those processors which evaluate a true predicate run the first arm of the conditional while the other processors wait, and then the tables are turned. All this is made intuitive by a combination of the local viewpoint and the *Rule of Block Synchronization*, which specifies that if execution starts at a C\* statement *S*, regardless of how processors may split apart inside *S*, all processors will have finished executing *S* before execution reaches any successor of *S*. This rule applies recursively to substatements of *S*. The rule also applies if *S* is a function body: No processor returns until all are ready to return.

The rules about domain declarations, control flow, and compound assignment all support the following central principle of C\*: Code inside a domain has the same meaning, for a single processor, as the identical code outside any domain.<sup>2</sup> That principle means it is very easy to port C code to C\*: The first step is to wrap a domain declaration around the whole source file, and this in practice often results in a working program.

In C\*, active sets are created using the *selection statement*, which uses (and extends) the dot operator. The left-hand side of the dot is an expression which specifies the desired active set, while the right-hand side contains code which is run relative to that active set. The left-hand side is a domain type name in square brackets; this specifies that all processors of that domain are to be activated. The right-hand side can be either an expression or a statement; conventionally it is a compound statement, enclosed by curly braces. Here is an example:

```
[domain Card].{
    extern Permutation perfect_shuffle(void);
    shuffle( perfect_shuffle() );
}
```

#### 4. The Type System

Like C++, C\* makes significant extensions of its type system over C. C\* has member types; a value of member type "T D::" locates a datum of the base type "T" in every aggregate of type "D". In

<sup>2</sup> The As If Serial Rule and Rule of Block Synchronization govern the interactions of multiple processors.

C++, the only thing you can do with a member value is ask for its datum within a specific instance.<sup>3</sup> In C\*, member types are *first class*, in that they may participate directly in all the C operations which normally apply to numbers, pointers, etc. A domain member value has a meaning apart from any single domain instance: it refers to the parallel set of values held by that member in all processors in the active set.<sup>4</sup> In fact, there is no "parallel" type or type modifier per se in C\* except for domain member types. This gives a convenient way to declare parallel variables (that is, all domain members can be used that way) Conversely we get a notation for the selection of a single value from a parallel set: the value of "X" on processor "P" is just "P.X".

A domain member (hereinafter called just a "member") whose base type is arithmetic can be given as an argument to any arithmetic operator, and the appropriate parallel operation is performed. (The semantics are specified by the As If Serial Rule.) A member can have its address taken, and the result is called an *offset pointer*, because its value is an offset into processor-local memory, independent of any particular processor.<sup>5</sup>

As in C++, member functions exist. (Remember that in C\* any C declaration can find its way into a domain.) However, domain member functions are called, not on behalf of a single message recipient as in C++, but for an active set of domain instances. Parallel code runs in two places: To the right of the dot in a selection statement, and inside domain member functions.

In C, a pointer's type depends only on the data type of the *target* of the pointer. In C++, the target type of the pointer can be modified by the "const" attribute, which is essentially orthogonal to other type attributes. In C\*, the target type of the pointer is distinguished not only by its basic data type and its const-ness, but also by its address space. (The keywords "mono" and "poly" have the same syntactic class as "const".) Here are representative declarations for the different cases:

```
float mono* front_end_pointer;
    /* Normal C pointer into front end. */

float D::* offset_pointer;
    /* Pointer into local memory of domain D processors. */

domain D* processor_pointer;
    /* Pointer to an instance of domain D. */

float poly* poly_pointer;
    /* Pointer into some domain instance, at some offset. */

float mono poly* system_pointer;
    /* Pointer to a single float, anywhere. */
```

But there's more. The pointer types in the above example are distinguished by the target of the pointer; and in C and C++ there is little more to be said, since there is only one processor to hold the pointer. But in C\* the *source* of the pointer can also vary. If a pointer is held by the active set, it is a parallel value, and so fetching or storing through it causes parallel communication between the processors of the active set and the targets of the pointers. As we saw before, there can be collisions in this case.

Finally, C\* has other smaller extensions to make it easier to program in a language with a complex type structure. First of all, function declarations can be prototyped as in C++. Prototyping becomes important in direct proportion to the number of types in the language, and their

<sup>3</sup>You can also take its address. It's a rare C type which is not addressable.

<sup>4</sup>All values of a member share a common size and processor-local memory offset, across the active set of processors. On the abstract machine level, we call such a set of values a *stripe*.

<sup>5</sup>Offset pointers were developed independently by C\* and C++, at about the same time and for similar reasons. Their usefulness is the greater in C\* because of the need to reference parallel values. Their syntax in the two languages is identical; in general C\* hews to C++ syntax and semantics wherever possible.

interconvertibility. For example, functions can take any combination of scalar or parallel (mono or member) arguments, and those arguments are placed as appropriate on the front-end stack or on the per-processor stacks. But (as noted before) scalar values can promote implicitly to parallel; and it is often desirable to pass scalar arguments to functions expecting parallel ones. In the absence of argument type information in the declaration of a function F, explicit casts must be issued for *each call* of F that requires the scalar-to-parallel promotion.<sup>6</sup> This problem is akin to the int/double problem in C, except that in C\* it is far more pervasive. Here are some examples:

```
domain D;

Permutation D:: nth_next_card(/int D::*/);
    /*Passes ints on the processor-local stacks.*/
nth_next_card(mark);
    /*Parallel int passed correctly.*/
nth_next_card(0);
    /*Implicit int->int D:: lost.*/

void D:: increm(/int poly* D::*/);
    /*Passes poly pointers on the processor-local stacks.*/
increm(&next_card->mark);
    /*Parallel ptr passed correctly.*/
increm(&mark);
    /*Implicit int D::mono->int poly D:: lost.*/

double sin(/double*/);
    /*Passes a double precision real on front end stack.*/
sin(PI/2.0);
    /*Double passed correctly.*/
sin(1);
    /*Implicit int->double lost.*/
```

Other C\* features which make programming safer and saner are overloaded functions, and the typedef syntax. The keyword "typedef" is syntactically similar to "sizeof", except that it yields a type, rather than the size of a type. The typedef syntax can be used to control the complexity of type names in declarations, and it also allows creation of type-generic macros.

In C\* overloading is included so that functions like "sin" can take either parallel or scalar arguments, invoking the appropriate parallel or scalar algorithm.

## 5. Comparison with C++

The C\* language does not extend C++. C\* lacks those C++ features which support the creation of abstract data types, such as operator redefinition, guaranteed initialization, and privacy mechanisms. It is envisioned that C\* will eventually expand to include these features. It will be possible to create abstract data types which can be instantiated as either parallel or scalar data, whose member functions will accept either kind of operand.

C\* is a language for data parallel programming, and so it has many facilities unrelated to anything in C++. However, there are some deep similarities between the languages. Both languages attempt to supply flexible, general means for specification and manipulation of aggregate data types (classes, domains). In both cases, data and functions are placed "inside" a type template, which is then instantiated many times. Whether those instantiations are real processors (domain instances) or just storage in the scalar machine (classes) is immaterial. In particular, the domain mechanism could be extended to allow C++ data abstraction, or C++ classes could be linked across files, as with C\* domains. Less radically, C++ like C\* could allow definition of member functions not pre-declared in the class definition.

<sup>6</sup> This style is allowed, for reasons of compatibility.

As noted above, C\* elevates members of domains to the status of first-class objects, because they are taken to denote parallel values across the current active set. In C++, member types play a somewhat smaller role.

The C\* control flow model has no analogue in C++, of course. However, the major features of that machinery could be expressed in C++ if there were a way to overload the "if" and "while" syntaxes, and if call-by-name parameters were supported. For example, the "if" construct would have an overloading something like this:

```
void
operator if(int parallel pred, void thunk t_arm, void thunk f_arm)
{
    int parallel save;
    save_active_set(&save);
    reduce_active_set(pred);
    t_arm;
    restore_active_set(&save);
    reduce_active_set(!pred);
    f_arm;
    restore_active_set(&save);
}
```

The C\* dot operator has a few strange cases not found in C++. They are derived from a generalized notion of what dot means, to wit: To evaluate "X.Y", first evaluate "X", and remember it, as an lvalue if possible. Then evaluate "Y", within a scope augmented by names (typically bound to subparts of "X") as determined by the type of "X". If "Y" yields a value, that is the value of "X.Y". The selection statement described above is a special case of this, with "X" being a primitive active-set constructor, and "Y" being a compound statement. The language also supports (although the current compiler does not implement) a powerful tool called *indirect evaluation*: If, in "P->Y", "P" is a parallel set of processor pointers, and "Y" is an arbitrary expression, "Y" is evaluated in an active set created by shifting the current set along the pointers "P". This behavior falls out as a natural consequence of 's general view of the dot operator.

C\* has a model of function overloading that is both simple and powerful. A set of overloadings can include both non-member and member (scalar and parallel) functions. That is, an overloaded function can span multiple scopes. This generality is required to support generic math functions like *sin*, which must be prepared to take either parallel or scalar arguments. When matching an overloaded function to actual arguments, the unique *best fit* is found. The definition of "best fit" is derived quite simply from the type conversion rules, which are taken to define a pre-order (i.e., transitive relation) over all types. The overloadings of a single function are ordered using the product of the conversion pre-order over the formal argument types. For a given set of actual arguments, an overloading is admissible if the actuals all can convert to the formals of the overloading, and the function call is made with the unique admissible overloading whose argument types are smaller (narrower) than all other admissible overloadings. (In the case of multiple, incomparable matches, the order of function declaration is consulted, with a compiler warning.)

## 6. Examples

Here are some examples showing C\* in action. The Sieve of Eratosthenes is a classic demonstration of parallel computation. It uses a process of elimination to isolate all primes smaller than a fixed number.

```
const NPROCS = 1000;

domain sieve {
    int is_prime;
} sieve_procs[NPROCS];

void
```

```

sieve::find_primes()
    /* show all primes < N active processors */
{
    poly int number = this - sieve_procs;
    /* Processor pointer subtraction. */
    poly int is_candidate = (number >= 2);
    is_prime = 0;
    while (is_candidate) {
        mono int next_prime = (<?= number);
        sieve_procs[next_prime].is_prime = 1;
        printf("%d\n", next_prime);
        /* Sieve out next_prime and multiples: */
        if (number % next_prime == 0)
            is_candidate = 0;
    }
}

void
main()
{
    [domain sieve].{
        find_primes();
    }
}

```

This code uses the previous example declarations for the "Card" domain. It shows how permutations might be generated and used.

```

domain Card*
Card::perfect_shuffle()
{
    /* Each Card returns a pointer to its successor */
    /* in the perfect shuffle permutation. */
    int n = this - deck;
    int dest = (n < NCARDS/2) ? 2*n : 2*n-NCARDS+1;
    return &deck[dest];
}

void
Card::shuffle(domain Card* dest)
{
    /* Given a permutation, move around card values. */
    if (dest != this) {
        dest->suit = suit;
        dest->value = value;
        dest->mark = mark;
    }
}

void
Card::ace_to_bottom_of_deck()
{
    poly int my_pos = this - deck;
    /* Choose the ace closest to the bottom of the deck: */
    mono int ace_pos = >?(value == 1) ? my_pos : 0;
    shuffle((my_pos < ace_pos) ? this

```

```
: (my_pos == ace_pos) ? &deck[NCARDS-1]  
: this-1 );
```

```
}
```

## 7. Conclusions

The C\* language is an effective tool for data parallel programming. It is about as "close to the hardware" as C or C++, yet provides a simple, symmetric, high-level model of computation. The central principle is that the programmer can create many processors, of many types, but they are all full-fledged C machines.

C\* and C++ need to solve similar problems in the management of complex data structures and type systems, so it is not surprising that similar mechanisms of aggregate definition, member functions, scoping rules, and overloading are found to be useful.

There is a persistent suspicion that parallel programming and object-oriented programming share common roots. Certainly in both worlds the concern is with specifying small data structures with local interactions. The success of C\* as a parallel language can only add to that suspicion, and some day we may have a "C++" in which to confirm or disprove it.

## 8. Acknowledgements

The C\* language was designed by the John Rose, Guy Steele, and Stephen Wolfram. John Rose wrote the C\* compiler for the Connection Machine system, with help from Sam Kendall, Guy Steele, Skef Wholey, Marshall Isman, and J.P. Massar.

Respectful thanks are given to the designers of the C languages which have preceded C\*: Dennis Ritchie and Bjarne Stroustrup.

Thanks to the Thinking Machines Corporation Software Group for their helpful criticisms of this paper.

# Implementing a Compiler in C++: Experience and Generalizations

John R. Rose  
Thinking Machines Corporation  
245 First St.  
Cambridge, MA 02142

(617) 876-1111  
rose@think.com, ihnp4!think!rose

December 7, 1987

## 1. Introduction

C\* is an extension of the C language which supports programming in the data parallel style on the Connection Machine system. Its principal extension over C is the introduction of aggregate types, called "domains", which represent processors, and the elevation of members of those domains to the status of first class types. C\* is currently being used to program Connection Machine systems. See [Steele] or [Rose] for more details on the language and its model of parallelism.

C\* is a new and complex language. It will grow as data parallel programming becomes more fully understood. And it is targeted to a new architecture which is itself changing. For all these reasons, it is important that the coding of the language's compiler be done in a flexible and extensible manner; exploratory methods of programming should be possible, so that experimentation can be carried out on the frontiers of the language and the target system.

The language's first implementation (for the Connection Machine system) is like that of C++: It uses a source-to-source translator which lowers a C\* program into an equivalent C program. For simplicity, we made the translator to be conservative, in that it leaves plain C programs relatively unchanged. Other C\* code, notably expressions manipulating parallel values, must be translated to sequences of macrocode-level primitives, e.g., two-address signed additions. These primitives show up in the output C code as calls to a library of Connection Machine primitives. Thus, the translator's intermediate language must be able to express a fluid mixture of source-level and macrocode-level operations.

In order to accomplish these goals, an open-ended, object oriented architecture was chosen for the translator, with C++ as the implementation language, and the Symbolics 3600 Lisp Machine as a development engine.

The purpose of this paper is to describe the way the translator's code shaped up, in regard to its use of abstract data types and objects, and to draw out some conclusions about use of the C++ type-definition facilities. First we will describe the overall structure of the C\* translator. Then we give an account of the C++ types used in this program, showing how they divide into two major type "kingdoms". This division is examined and rationalized. Next, we will say a few words about the development environment, which is a mixture of C++ and Lisp. We remark on our experience debugging with user-defined types. Finally, we briefly discuss our approach (within the development of the C\* translator) to problems of boilerplating, header-file control, separate compilation, and type instantiation.

## 2. Translator Overview

The intermediate representation manipulated by the translator is essentially a C\* abstract syntax tree. Each C\* language construct is supported by one or more C++ object types. The main activity of the translator (besides type-checking) is iteratively transforming C\* tree nodes into equivalent but lower-level C\* constructs. This activity continues until the program uses only the constructs of the C language. Consequently, the translator has a very simple pass structure, but complex data structures. C-level concepts, such as type and expression, are always represented; there is never a point at which

---

Connection Machine is a registered trademark, and C\* is a trademark, of Thinking Machines Corporation. VAX and Ultrix are trademarks of Digital Equipment Corporation. "Symbolics 3600" is a trademark of Symbolics, Inc. Zeta-C is a trademark of Zeta-Soft, Inc.

they are lowered to a radically simpler notation such as Register Transfers (e.g., [Davidson]).

### 2.1. Phase Structure

The translator's phase structure is as follows. A parser builds a tree representation of the program from the bottom up. The flow of top-down information (notably, scoping of identifiers) is controlled by a uniform "context" module. As each sub-expression is built, it is type-checked. The lowering process is then carried out, iteratively. As each function is completed, a separate pass inserts code to manage parallel threads of control. Next, the function's Connection Machine stack frame is built. Finally, C code is emitted for each compilation unit; this is done by a process inverse to the original parse. The results are handed to the system C compiler.

### 2.2. Messages and Objects

All phases of the compilation process are invoked by sending messages to the code tree. The C++ virtual function mechanism is used to allow each of the many kinds of tree nodes to perform its peculiar computation. Each object type may have its own handler for any of those messages. For example, the "Expr:: resolve\_type" message has 27 handlers.

Because the intermediate language is quite rich, there are many node types to keep track of. The virtual function mechanism makes that complexity manageable, because node types can be restricted to be defined and referenced by a small piece of code. This code can be understood solely in terms of its interaction with the well-known interface of the base type.

For example, the `String_constant_expr` node type, which encodes a C string literal, is defined and referenced solely within a span of 64 lines of one file. The constructor function `make_string_constant_expr` returns (a pointer to) an `Expr` node, `Expr` being the well-known type which defines the interface to all expression nodes. Therefore, callers of `make_string_constant_expr` don't have to know about the `String_constant_expr` type. The message handlers for this type support type conversion and output. Properties of the node related to its compile-time constancy are realized by message handlers in a base type called `Constant_expr`; this allows algorithms to be shared by integral and floating-point constants as well as strings.

### 2.3. Phases and Types

The phase structure of the translator tells only half of the story: The *type structure* is the other half. The phase structure is characterized by the various messages sent (e.g., "resolve\_type", "emit\_expr"), and the ordering of those messages. The type structure is dual to this, and is determined by the meaning of each node type, and (consequently) its action in response to each message. An important simplifying principle is that the meaning of each tree node is independent of time. Put another way, the translator uses just one internal representation for all phases.

In a typical compiler, data structures are phase-dependent, either because each phase *translates* one internal notation into another (e.g., expressions to quadruples), or because a phase *decorates* a data structure with additional information (e.g., expression type), which only later phases may access. Translation and decoration are useful paradigms, but they are done in a time-invariant way by the C\* translator.

The process of lowering C\* code to plain C code is a translation, but it is done on one node at a time, replacing that node with an equivalent, lowered node. No new representation of the program is created, but an existing one is modified. The meaning of each expression is maintained as an invariant. For example, the expression " $(x+y)+z$ " may, at some point, be represented by an unlowered addition of " $z$ " and a the lowering of the sum of " $x$ " and " $y$ ". That lowering turns out to be a temporary allocation, an assignment, and a two-address Connection Machine addition. The meaning of the top-level addition does not depend on assurances about the lowering of its subexpressions, nor is their consistency impaired by the fact that their parent node is still unlowered. Thus, translation is expressed as an incremental process of tree surgery.

An example of data structure decoration in the C\* translator would be the allocation of Connection Machine stack frames to code blocks. Early phases cannot use stack frame information, before all

temporary quantities have been allocated. How then do we understand the nature of code blocks apart from the sequencing of the storage-binding phase? The key is to regard the decoration process as the lazy evaluation of an attribute of the code tree. The account of a code block's meaning includes a description of its stack frame instance variable, but the computation of this instance variable is delayed until the storage requirements of all sub-blocks are known. If a sub-block subsequently attempted to allocate more stack frame storage, a compiler bug would be detected and signalled.

Localized propagation of attributes through a parse tree is a well-understood phenomenon studied under the name of "attribute grammars". (See, e.g., [Koskimies].) In evaluating a set of attributes, the phase structure is derived from and subservient to information flow within the parse tree data structure. In an object-oriented system like ours, that information flow is determined individually by the various node types. Attribute grammar theory gives ways to determine statically an optimal evaluation ordering for the attributes. Our solution to the ordering problem is to use lazy evaluation, and let the phase ordering be shaped by the dynamic execution of the various lazy computations. For example, a code block's stack frame attribute is just an instance variable, but its accessor function makes sure that the first request for the stack frame triggers its computation. This in turn generates requests for the stack frames of sub-blocks, and the whole effect is of a postfix traversal of the code block tree, driven only by the requirements of data flow.

This technique uses more runtime computation, and so is slower than static methods. But like all runtime checking, it is robust, even in the face of a rapidly-changing program. An important part of the debugging of our phase structure was tending to errors caught by these runtime checks.

## 2.4. Type Structure

The principal node types of the translator are as follows. During the parse, trees of C\* code are built out of Expr and Block nodes. As one would expect from the nature of C, Expr nodes denote value-producing computations, while Block nodes express storage allocation and complex control flow. Unlike C, some Expr nodes can contain Block nodes, but the lowering process sets that aright. During the parse, top-down information is maintained by a stack of Context nodes, which are responsible for the scoping of names. Context-dependent control flow primitives like "break" rely on the scoping of special label names.

Whenever a name is bound, or an object is otherwise created, a Decl node records the fact. The translator has no need for a specialized symbol table module. During the parse, the Context stack mediates the translation between names and Decl nodes, and thereafter, all names are ignored, until it's time to unparse.

The C\* language has a complex type system, which provides for parallel values, processor aggregates, and multiple address spaces. About one quarter of the compiler's code is devoted to keeping track of the C\* types and their interrelationships. A type is represented by a Type node, and the method for converting from one type to another is recorded in a Conversion node. Each type has a characteristic representation, one facet of which is the plain C type which is used to carry that representation in the translated program. A Type node is responsible for creating and allocating objects of that type; therefore, Type nodes mediate the creation of Decl nodes. Decl nodes, in turn, mediate the creation of Expr nodes.

## 2.5. Value-Oriented Programming

A final word remains to be said in this overview of the C\* translator. The programming style does not rely on the ordering of side effects on global variables or data structures. The action of a function call is characterized only by the computation of its return value, which is based only on its arguments. The Context stack is the notable exception; name lookup is always performed relative to the current state of the parse.

As discussed above, side effects are applied to data structures in a localized manner. There is usually a level of abstraction above which the mutations undergone by a node have no effect on its meaning. From this high level, we can view the program as side-effect free. This is an important simplification, because it lets one understand functions on a call-by-call basis, without reference to the past history of the program's execution.

Like any C program, ours is packed full of side effects. However, the side effects are tightly controlled, by the following methods:

- hiding them behind a lazy-evaluation abstraction,
- making them idempotent in other ways than lazy evaluation,
- synchronizing them with flag bits,
- ensuring that they preserve some high-level invariant,
- seeing that they take effect in only one place,
- first copying the data structures they affect.

The result is a program which is simpler, more reliable, higher-level, and easier to decompose for testing.

Programming with such close control of side effects could be called *value-oriented*, since the emphasis is on abstract values with no internal state. Such a style is complementary, rather than contradictory, to object-oriented programming. Stateless objects can be built out of state-full components. For example, consider a recursive Fibonacci function which caches its results, or the operation of the Lisp INTERN function.

The C\* translator uses value-oriented programming in its manipulation of lower-level data structures. If a function is to produce more than one return value, the standard C method of storing extra values in a global variable is rejected in favor of returning a structure of all values together. This is made easy in C++ by a parametrized type called Values(). Global variables are also used to store function arguments. Suppose a closure of a function with *free variables* in its body is to be produced, to hand to a higher-order function like qsort(). The standard C method is to set aside global value cells for the free variables, and so there can be only one instantiation of the closure active at a time. The C\* translator makes more complex use of closures than that, and so once again the global variables are discarded, in favor of a parametrized C++ type called Closure(), which simulates function types.

Many algorithms deal with sequences of values, and some high-level languages provide primitive sequence types, often linked lists. Programmers tend to use lists in value-oriented ways, even though the abstraction supports shared side-effects. In Lisp, there is a performance/robustness tradeoff: Use APPEND and your program will run slowly, or use NCONC and your list values may be corrupted by unexpected side effects.

With these issues in mind, we designed a parametrized type Seq(T) which stores sequences of values of some base type T. It has the flexibility of linked lists: Prepend and append are cheap operations. The abstraction offers complete insulation from shared side effects. Just as setting a bit in an integer variable has no effect on any other integer value, changing an element of a sequence variable affects no other sequence value. Storage is shared when possible, and reference counts are used both to recover unused storage and to ensure that changes to a sequence are not detectable by another party.

Below is a terse list of the value-like types which we found useful in building our software. These types show up everywhere the C primitive types are used: as global, local, and instance variables, as function parameters and return values, and as sources and destinations of assignments. The third column gives a Common Lisp function corresponding to the type. (See [Lisp].) For example, the Map() parametrized type implements a sequence of ordered pairs, defining a finite function, on which lookups can be performed. This is (essentially) the same abstraction as the Lisp A-list, which is accessed by the ASSOC function. Likewise, Lisp functions use the VALUES primitive to return multiple values.

### 3. Two Type 'Kingdoms'

In fact, the development of the C\* translator required the use of two wholly different kinds or "kingdoms" of types: The low-level, generally useful, value-oriented types described in the previous subsection, versus the high-level, application specific, object-oriented types around which the translator is organized. These two kinds differ in nearly every way, even though the same programmer designed them. It is a measure of the expressiveness of C++ that both kingdoms could be realized in that language. Below we display a tabular comparison of the two kingdoms. We call the low-level types *value oriented types*. For a treatment of "data abstraction" versus "object-oriented programming", see

*Value-Like Types*

Facility	C++ name	Lisp function
multiple return values	Values(T...)	VALUES
uniquified strings	Symbol	INTERN
value sequences	Seq(T)	LIST
finite mappings	Map(T,U)	ASSOC
ref-counted storage	Chunk	CONS
nested functions	Closure(T,U...)	LAMBDA
large integers	Bignum	PLUS
streams	Stream	OPEN

[Stroustrup], which treats the latter as an enrichment of the former. Our "value oriented" types are built by enriching the basic data abstraction paradigm in the area of types parametrization, to support value-oriented programming. Thus, both object-oriented and value-oriented types enrich the basic idea of an abstract data type. (In what follows, we will abbreviate "value-oriented" as VO, and "object oriented" as OO.)

We now proceed with a point-by-point comparison of VO types with OO types, as found in the C\* translator, hereinafter referred to as "the application". The meaning and behavior of a VO type is

*Comparing the Kingdoms*

paradigm:	Values	Objects
level	low	high
specification	algebraic, axiomatic	open-ended, functional
source	in a library	in application code
declaration	as object	as pointer
usage	immediate dot, C operators	through pointer, message sending
derived types	well known	hidden, easily created
base type	private	public
new types	via generic instantiation	via derivation, inheritance
why inheritance?	reusing untyped code	common interface
definition	automatic, if used	class generation tool
Cosmic Type	never	yes
virtual fns	rarely	always
size	small, fixed	variable
allocation	stack, some heap	always heap
storage mgmt	ref. counts	by hand
side effects	no sharing, pure values	yes, shared structures, all copying explicit
C++ weakness	type params	boilerplate

defined by a small task which it performs. Often, that task can be described algebraically, or by a small set of axioms or invariants. Operator overloading may be used to reflect the algebra. By contrast, an OO type has an open-ended meaning, usually involving many relationships to other modules and types in the application. The interface an OO type never uses operators, but rather member functions, which are thought of as symbolic *messages*. Because of their simplicity, the VO types are reusable, and belong in libraries. The OO types are defined in the midst of the application's code.

When a VO value is declared, the type name used refers to the abstract type itself; there is no reason to involve any other type. However, when an OO value is declared, the *pointer* type is invariably used to hold the value. This is because application code almost never knows the exact class that the value will have at runtime, and in fact the class will generally vary from moment to moment. For this reason, OO type identifiers such as "Expr" do not refer to a C++ class type, but rather to its corresponding pointer type. The macro-function call "Class(Expr)" yields the actual class type, which is never used except in the actual definition of the "Expr" type. We often say Expr:: resolve\_type where Class(Expr):: resolve\_type is correct.

Thus, all uses of OO types are "pointer-like". Sending a message uses the arrow syntax (e.g., "expr->resolve\_type()") rather than the structure dot syntax. Even accesses to data fields use functional syntax "expr->type()" because use of the non-functional syntax ("expr->type") requires a commitment to Expr representation. For example, that would preclude lazy evaluation of the Expr:: type field. The arrow syntax is therefore to be read as message sending. The VO types are not pointers, so they are accessed via the dot syntax. Pointers to VO types shouldn't be necessary often, since they support call-by-value, but if such pointers are to be indirected, the star syntax should be used. This avoids confusion with the main use of the arrow operator, which is to express OO message passing.

There is a striking difference between VO and OO types in their use of the C++ class derivation mechanism. With VO types, the derived class is well-known, and the base class is private, while with OO types, the base class is public, and the derived classes are hidden. As we saw above with the String\_constant\_expr node type, very little code refers to the derived type, and all of its interaction with the rest of the system is mediated by the messages of the base class Expr. Now, Expr is made public in String\_constant\_expr, so that a newly-created pointer of the latter type can be immediately changed to the former type. Note that because the code for the derived class is so localized, it is very, very easy to create and manage many such classes. (The C\* translator has over 100, which is perhaps excessive.) So, OO types use inheritance as a means of attaching many different data structures to one interface.

VO type do not change types like this. The full derived type is visible at all usage points. What is invisible is the base type (if any; you can't even tell). The reason for using the C\* derivation mechanism in this case is to share code between related VO types. For example, the parametrized Seq(T) type inherits from an untyped private class called "anySeq", which knows about sequences-oriented storage management, and that in turn inherits from the Chunk class, which handles all reference-counting. In both the VO and OO cases, classe derivation is used to separate interface from implementation. In the OO case, the base class carries the interface, but in the VO case, the derived class does. A VO derived class often adds type checking to an untyped implementation in the base class, whereas an OO class adds an implementation to an interface. Often, the interface functions defined in an OO base class must be redefined by all derived classes; they may signal an error if called directly. But in both cases, the base class is "abstract", in the sense that it is rarely instantiated.

What are the degrees of freedom by which types may vary? In the VO case, they are preset by the abstraction, as parameters of the type. The Seq(T) abstraction has one, and the Values(...) abstraction has many. Creation of a new VO type such as these is done ideally just by using the VO type with a new set of parameters. The system should take care of instantiating it. (This is the case with the C\* development system, as we'll see below.) But with OO types, creation of new types is done by derivation, as we saw above.

Because of their sharing of interfaces, OO types admit the existence of a universal base type, to which all OO types can be converted. (We call it "AnyClass".) This "Cosmic Type" does not have any application-specific properties. Rather, it supplies certain useful services, relating to input/output, dynamic type checking, storage allocation, debugging, and the like. These tasks recur in all large OO systems. The Cosmic Type defines virtual functions to perform them, and each subtype redefines them

for itself. For example, the "copy\_bits\_v" message, defined by anyClass, allocates a new block of storage and performs a memberwise initialization of it from the recipient object. Here is a partial list of application-independent operations a Cosmic Type might support:

- initialization (handled by C++)
- destruction (handled by C++)
- assignment (now handled by C++)
- equality, hash code creation
- ordering (lexicographic, componentwise)
- relocation
- copying
- dynamic type-checking
- census-taking
- debugging, inspection
- runtime sanity-checking
- printing
- disk save/restore, persistency
- XDR handler generation

The definitions of these operations will usually distribute the operation over an object's members and base type.

VO types should probably not partake of the Cosmic Type (unless there is a prior goal to make the language a COOL: Completely Object-Oriented Language). This is because use of virtual functions requires a storage overhead of one word, and most VO types in fact consist of one or two words. This means a large proportionate cost in space and time for features which are needed mostly to deal with complex objects. (Would you really require ints to be COOL? No, and VO types are in the same category.)

The problem with the Cosmic type is that it requires each new derived type to generate highly stylized code to implement each Cosmic operation. The size of this code is proportional to the product of the number of instance variables defined by the new type, times the number of Cosmic operations. Multiply that by the number of OO types in the cosmos, and you have a heavy burden on the programmer. This kind of redundant code is called "boilerplate", and it is the kind of boring, error-prone code programmers are worst at. A computer should manage it. (Ours does, but our solution is non-general.)

As noted before, VO types rarely use virtual functions. This is because they use inheritance sparingly, and because their size makes the overhead significant. On the other hand, OO types *always* use virtual functions, because they are the fundamental mechanism by which new message handlers are linked into the existing interface. In fact, there is almost never a reason to use non-virtual functions, when multiple handlers for a message exist. When only a single handler exists, good performance dictates use of non-virtual linkage, but that is only an optimization which does not change the program's behavior. So it is safe to say that, apart from that optimization, *all* OO members functions are virtual. That is in keeping with their interpretation as messages. (One of the uses of our boilerplate generator is to make a final decision on virtual vs. non-virtual.)

VO types tend to be small, only a word or two. This allows them to be passed around inexpensively. OO types are also small: A pointer takes up just one word. But the object pointed to can be quite large, and the size varies from object to object. Therefore, OO values are always allocated on the heap. In contrast, a VO value, like an int or float value, is usually put on the stack (or the static data segment). If the value has a variable information content (like Seq(T)), part will be placed on the heap, but that fact is irrelevant to the type's usage, since the management of the heap storage is completely internal to the VO type. The storage of OO values must be managed by the programmer, with the "delete" operator, since normal C pointers are used to access them.

Likewise, the programmer is responsible for managing the storage sharing of OO values, and the resulting communication of shared side effects. In fact, if a fresh copy of an object is desired (to free it from subsequent side effects to the original) an explicit copying operation must be performed. In practice, this is not often done; OO types tend to have well-defined "identities", which copying does

violence to. The VO types are value-oriented, so they have no detectable identity. It is a bug if an update to one value causes a change in another, even if the values were identical.

Note, finally, a similarity: Both VO and OO types stretch the limits of the C++ language. The VO types gain their richness from parametrization, which is currently poorly supported. The OO types gain their richness from the interaction of a centralized interface specification with a large number of derived classes, but maintaining large numbers of redundant declarations is difficult to do by hand.

#### 4. Development Strategy

The initial development of the C\* translator was performed on a Symbolics 3600. The programming environment on that machine is one of the best available. It offers an integrated editor, incremental compilation and execution, dynamic type- and safety-checking, automatic storage management, the best debugger in the world, and of course full support for the Lisp language. Having a Lisp system available meant we could defer implementing the value-oriented types, since they are supplied as primitives in Lisp. The C language was available for that environment, and it was well integrated. (The product was Zeta-C; Symbolics has since released their own C also.)

The incremental features of the Lisp machine allow the developer to compile and test the program as each function is written. There is no turnaround time, such as that spent by traditional systems in compiling and linking large chunks of code. Test code can be written alongside the program unit being tested, and invoking the test consists of calling it from the C interpreter. This can be done after every change, if desired.

Thus, the Lisp Machine is an ideal environment for prototyping, exploratory programming, and rapid development. However, the program was targeted to the DEC VAX running Ultrix, serving as a front end to the Connection Machine computer. Thus, the reliance on Lisp could not continue past the initial stages of development. The goal was to program in such a way that the same source code developed on the Lisp Machine could be compiled and run on the VAX.

This goal was accomplished by carefully considering all abstract types, and giving them appropriate, equivalent implementations on both the VAX and the Lisp Machine. In particular, our object-oriented types were implemented using the Lisp Machine's Flavors system. The value-oriented types were just "syntactic sugar" for various useful Lisp constructs, such as symbols and A-lists. A small modification was made to the Lisp Machine's C compiler, so that new abstract types could be introduced from the Lisp world, and used in C programs with the dot and arrow operators. Note that there was no need to extend the C language to allow the *definition* of such types. Specialized Lisp code creates the proper data structures within the compiler, without help from any C source file. (It is interesting to note that most of C++'s syntax extensions are to declaration syntax. C++ expressions and statements are solidly C-like.)

The implementation of each value-oriented type consists of a piece of Lisp code which declares the type to the C compiler. Likewise, the implementation of each object-oriented type consists of a Lisp form describing the fields and functions of the type. The form is called DEF-C-CLASS, and is much like DEFSTRUCT or DEFFLAVOR. The C source code defines the out-of-line member functions.

On the VAX side, all user-defined types are implemented as C++ classes. Each object-oriented type is defined by its own header file, which is generated by a Lisp program which analyzes the defining DEF-C-CLASS form. There is a rather large semantic gap between the Lisp Machine and the VAX: In the former environment, the value-oriented types were essentially primitive, but on the VAX a fair amount of library code had to be written to support Seq() and the rest.

One problem with debuggers and user-defined types is that they rarely recognize one another. In the case of our value-oriented types, the question was dodged, because the Lisp Machine in fact supports them as primitives. That is, a Seq(Int) value prints out as a list of numbers, a Symbol value prints as a word, and so on. Also, there is a way for Flavor types to define their own printed representation, so the object-oriented types were handled that way. Why is printing important? Because all debugger information displays convey their information by printing relevant program values: Values are printed in stack frames, function call traces, variable cell traces, source-level print statements, and backtraces.

Often there is no further pause, during which a programmer could ask for further display of substructures. In such cases, the difference between an octal constant and a pretty-printing is great indeed.

These problems of printing are more acute in the VAX debugger, which knows about a relatively small set of types. In particular, all object-oriented types are implemented as pointers, and so print as hexadecimal constants. The present implementation of C++ forces most value-oriented types to be passed by reference, and so they too are printed as pointers into the stack. Such hexadecimal constants must be fed manually to a print function, when a convenient moment arises, by which time the data structure has often changed irrevocably.

Other useful debugging operations are printing out an object's substructures and modifying them. Again, the user-defined type should participate by defining protocols for browsing and modifying objects of that type. On both the Lisp Machine and VAX, the system debuggers have enough smarts to identify the substructures of our various types.

#### 4.1. Efficient Separate Compilation

One difference between the Lisp Machine C and most other C implementations is that compilation of a source file proceeds in an environment which includes all objects declared in all previously-loaded files. Therefore, the information in a header file (or a Lisp type definitions) need only be read once, and subsequent file compilations or incremental compilations can proceed immediately with full benefit of the loaded declarations. (This is a simplified account; there are a few mechanisms for protecting programs from unintended interaction.) In particular, the usual practice of forward-declaring external functions before they are used may be disregarded.

The equivalent practice on the VAX would be to include, at the top of every source file, a massive header file which would contain forward declarations of every externally-linked function, as well as definitions of all classes. Such a compilation would not terminate in a reasonable amount of time, making the equivalency a moot point. This problem of large header files is getting more and more serious in the C++ community, for two reasons: First, function calling sequences are getting more and more specialized, because of the richer type structure, and hence forward declarations are now required. Second, as abstract types get more frequent and more complex, a larger proportion of a given program will consist of its header files, and each header file must be compiled once for each source file. Separate compilation may soon be just as important for header files as for source files.

Our solution to this problem is partial, but it does make the difference between feasibility and the contrary. A program called the "Source Linker" scans a source file just before it is compiled, analyzes the symbols it finds therein, and generates a header file which defines the appropriate classes and external functions. A topological sort puts the declarations and inclusions in the proper order. The Source Linker filters out the needful 2000 or 3000 declaration lines out of a possible 14000 or so; this keeps the header files within reasonable bounds. Almost all of the header file volume is due to the declarations of the object-oriented types. If they were made less interrelated, the Source Linker would be able to generate smaller headers. Also, the top-level object-oriented classes currently contain many unnecessary interface functions; this is due to the great ease of introducing new messages in the Lisp Machine Flavors system: Flavors messages are not pre-declared as C++ member functions must be.

#### 4.2. Instantiation of Parametrized Types

The Source Linker is responsible for detecting instantiations of parametrized types, and issuing forward declarations for them. As illustrated in the example below, at each use of a parametrized type "Foo(Bar)", a forward declaration of the form "DeclareFoo(Bar)" must be visible. Also, at exactly one place in the program the defining form "DefineFoo(Bar)" must be compiled.

A different invocation of the Source Linker analyzes the final object file, and generates those per-program definitions, as well as definitions of statically-referenced Symbols, and virtual function tables.

#### 4.3. Boilerplate

The DEF-C-CLASS form generates boilerplate code in a number of ways. Most importantly, it allows omission of the type of a member function, if that function is a redefinition, in which case it

takes the type from the base class. For example, here is the definition of `String_constant_expr`:

```
(def-c-class string-constant-expr constant-expr ()
  (value chunk)
  (EXTERN-MEMBER bash-type)
  (EXTERN-MEMBER emit-expr)
  (EXTERN-MEMBER put-self))
```

Here, `EXTERN-MEMBER` is simply a macro which expands to a forward declaration of the given name as an untyped function. The C++ header file generated from this code includes the declarations of one data and three function members, and also the following items:

- A functional interface for the member datum:  
`Chunk& value(){ return _value; }`
- A 3-argument constructor, which was requested by the `Constant_expr` class to be provided for every derived class. One of the arguments is the `VALUE` data member, which is a `Chunk` of string characters.
- A "ClassDescription" structure which will supply runtime typing information about this type.
- The virtual function `String_constant_expr::class_description`, which returns a pointer to the `ClassDescription`.
- The virtual function `String_constant_expr::copy_bits_v`, which returns a componentwise copy of a `String_constant_expr`, typed as an `AnyClass`.
- The non-virtual function `String_constant_expr::copy_bits`, which casts the result of `copy_bits_v` to a `String_constant_expr`.
- A constructor for use by the `copy_bits` message, which accepts a reference to the object being copied.

In many cases, a base class defines a virtual function as a hook for derived classes to use, and a trivial definition is needed for the base class. This trivial definition either yields a null value of the appropriate return type, or signals an error. Both cases are handled by a macro called `STUB-MEMBER`.

Other macros implement lazily-evaluated instance variables, message delegation, and accessors for read-only instance variables, which return plain values instead of references.

There is a macro `NEW-MEMBER` which defines a constructor; its arguments are the names of instance variables which are to be initialized by corresponding constructor arguments. A related type of boilerplate is the "setup" message, defined by the `SETUP-MEMBER` macro. It assigns values from arguments to named instance variables. Setup messages can be cascaded, because they return the self pointer "this".

There is a boilerplate inheritance mechanism, by which a base class can insert arbitrary code into all derived classes. This is used in several places, usually to define a constructor uniformly across a set of classes. This is the way both of `String_constant_expr`'s constructors were defined: The bitwise copy constructor was requested by the `AnyClass` base class, and the other constructor came from the immediate base class `Constant_expr`.

Our boilerplate is generated by Lisp code, which is clearly not suitable for inclusion in a C++ compiler. However, based on our experience, it seems clear that C++ must one day address these issues.

## 5. Conclusions and Warnings

Object-oriented programming is useful for managing the code of a language processor, if complex, open-ended data structures are required, as in our case. However, because of the abstract nature of the node types, our compiler has trouble performing pattern matching on complexes of more than one node. Therefore, it does not perform many optimizations, and the complex transformations it does make tend to overreach abstraction boundaries, although type-checking is never circumvented.

The ability of the Lisp Machine to handle complex type systems encouraged us to use many types in our object-oriented data structures. When a new conceptual entity presented itself, it was the work of a moment to define a type for it. Adding new operations to existing interfaces was also easy. All this was so easy that we added many types and messages which were not really very important. The resulting bloat has had three effects.

First, the translator is slow. The large number of messages used by the object-oriented types results in many function calls at run-time. Functional abstraction is expensive! A factor which multiplies the expense of function calls is the reference-counted storage management of the value-oriented types. Every time such a value is passed as an argument, reference counts must be updated. A further factor is the C++ member-function calling sequence, which takes the address of the object; this prevents many optimizations from taking place in code which uses our value-oriented types.

A second consequence of our over-large object-oriented type system is that non-incremental compilations are expensive, on either the VAX or the Lisp Machine. Thirdly, and of most importance, the code is difficult for new programmers to learn, since the work of sorting the conceptual wheat from the chaff has not been done.

The lesson of those last points is that the experimental, exploratory programming methods allowed by environments like the Lisp Machine do not provide a substitute for careful design. Decisions made in the heat of coding should always be reviewed and reconsidered later. Such reconsideration must be included in the overall cost of programming. When a program is produced incrementally, initial results come quickly, but the code usually needs rewriting.

Nor is the reverse of the previous lesson true: Careful design is no substitute for interactive, incremental programming. With a good overall design directing the coding, productivity in the Lisp Machine environment is usually several times as great as that in the VAX environment. This has been our experience with the C\* translator; one afternoon on a Lisp Machine can get as much done as a week with the VAX.

Not only coding but also the design process itself can be helped by exploratory programming. Many small decisions are made most appropriately in the heat of coding, when all the relevant local factors are fresh in the programmer's mind. Also, their consequences can be immediately evaluated by an experiment invocation of the new code. If any small decision has a non-local effect, such as the addition of a new data-type or message to the system, a later re-evaluation is necessary. Maintaining a change log on all class definitions would help in this process.

We discussed the two "kingdoms" of types: Object-oriented and value-oriented. With respect to the previous discussion, it appears that the value-oriented types are the ones which are designed wholly in advance, and which have simple, application-independent properties.

The power of object-oriented types lies in their open-endedness. This also makes them dangerous to the simplicity of a system, unless the meaning of each message is carefully defined, and the number of messages restricted. Once again, careful design is needed, both before and after coding.

We have touched on a number of needs for further research on C++: Parametrization of types is needed, as are mechanisms for generating boilerplate code. Efficient debugging of C++ programs needs smart handling of user-defined types; debuggers probably need type extension mechanisms analogous to compilers. Finally, separate compilation of C++ class declarations is desirable.

## 6. References

- [Davidson] Jack W. Davidson & Christopher W. Fraser, "Code Selection through Object Code Optimization", *ACM Trans. on Programming Languages and Systems*, Vol. 6, No. 4, Oct. 1984, pp 505-526.
- [Koskimies] K. Koskimies, K.-J. Raiha, and M. Sarjakoski, "Compiler Construction Using Attribute Grammars", *Proc. ACM SIGPLAN Symp. Compiler Construction*, June 1982.
- [Lisp] Guy L. Steele L. Steele, Jr., *Common LISP: The Language*, Digital Press, Burlington, MA, 1984.

- [Rose] John R. Rose, "C\*: A C++-like Language for Data-Parallel Computation", this proceedings.
- [Steele] John R. Rose and Guy L. Steele, Jr. "C\*: An Extended C Language for Data Parallel Programming", *Proc. Second International Conf. on Supercomputing*, vol. 2, pp. 2-16, International Supercomputing Institute, Inc., St. Petersburg, Fla. (1987).
- [Stroustrup] Bjarne Stroustrup, "What is 'Object-Oriented Programming'?", *Proc. ECOOP*, Paris, June 1987.

## 7. Appendix: Code Example

Here is a small but complete program illustrating the definition and use of a abstract data type whose values are sequences of another type. These values behave much like Lisp lists, except for strong typing of the sequence elements, and protection from shared side effects.

```
#include <stream.h>
#include <Seq.h>
#define Int int
// Once per file:
DeclareSeq(Int)
DeclareSeq(Seq_Int)
main()
{
    Seq(Int) row;
    Seq(Seq_Int) res;
    for (Int i = 1; i <= 5; i++) {
        row.add(i);
        res.add(row);
    }
    cout << res << "\n";
    cout << "...should be:\n";
    cout << "((1) (1 2) (1 2 3) (1 2 3 4) (1 2 3 4 5))\n";
    // res is triangle, not a square, because
    // there is no detectable structure sharing
}

// Once per program (can be automatically generated):
#include "seq-internal.h"
DefineSeq(Int)
DefineSeq(Seq_Int)
```

## A Style for Writing C++ Classes

Peter A. Kirsulis

AT&T  
Denver, Colorado

### Extended Abstract

The ordering chosen for the elements in a C++ class definition can greatly impact a user's ability to understand the external interface of the class.

C++ class definitions contain not only specifications, but also some required implementation details. However, many of these details are of no interest to the user of the class, and should not have to be read to determine the operations which comprise the external interface of the class. Thus, the elements of the class should be ordered, as follows: place the declarations<sup>1</sup> (specifications) of member functions and friends needed by users of the class first; place the declarations of member functions needed by implementors of classes to be derived from this one next; and place the declarations of private member functions and the definitions of the data members of the class last. The definitions (code

1. A *declaration* associates a type with a name and makes it known within some scope, a *definition* associates a value with a name; thus, `class C`; and `int C::f(int)`; are declarations, while `class C { ... }`; and `int C::f(int v){...}` are definitions. (In C++ definitions can also be considered declarations, since the name-type association is also given; but for the purposes of this discussion, the introduction of a name will be termed a declaration only if the defining value (e.g. function code body) is not present, that is, it cannot also be a definition.)

#### File: C.h

```
// Specification (External)
class C {
    friends
public:
    constructor(s)
    destructor
    member-function-decls

    // Specification (Internal)
protected:
    constructor(s)
    member-function-decls
private:
    constructor(s)
    member-function-decls

    // Implementation
protected:
    data-members
private:
    data-members
};

inline member-function-definitions
```

#### File: C.c

```
#include "C.h"
member-function-definitions
(except inlines)
```

Figure 1: C++ Class Definition. All specification information needed by users of the class is placed first, with internal specifications and required implementation details following.

bodies) of all member functions of the class should follow the class definition, with those functions which are to be inline expanded preceded by the keyword *inline*. If the class definition is to be split between a header (".h") and code source (".c") files, the inline expanded member function definitions should follow the class definition in the header file. All other member function definitions should be placed in code source file(s), which reference the class definition through a preprocessor *#include* directive. Only the declarations of the member functions should appear in the *class* construct itself, to keep the implementation separate from the specification, reducing clutter. Figure 1 presents a class definition template which observes this ordering.

Note that only private and protected<sup>2</sup> data are shown in Figure 1. Public access to data is undesirable, since it defeats the purpose of data encapsulation and precludes data validation; it is not necessary for efficiency considerations, since C++ permits inline functions for data access and update. Special access needs that arise can be handled through the use of either *friends*, or protected data and derived classes, when direct access to the raw data is necessary. The non-member functions which need access to the data can obtain it, while general public access is still prevented.<sup>3</sup>

2. The *protected* designation makes the functions and data declared within its section accessible to classes derived from the class (behaving like *public* to derived classes), but prohibits access by client software which uses the class (behaving like *private* to non-member and non-friend functions).
3. For an unusual example of the use of protected data, consider a C++ library class which implements complex numbers and provides some operations on them, but does not provide some other operations deemed necessary. If these new routines are added to the complex number class, the original library compiled from this class will need to be regenerated. Instead, if the data elements are placed in the protected section of the class (a change to the .h file only), and these new routines are defined as member functions of a new class derived from this one (but containing no new data), then the new functions can directly access the data without requiring the original complex number class library to be recompiled. The problem of additional operations on complex numbers, given a C++ complex number class (written elsewhere) arose in the work of Jon Sauer of AT&T. He opted to

Functions classified as *friends* of a class should be declared as part of the external specification of the class, although whether at the beginning or end of the section is not clear. The decision may rest upon the number of friends present, although a choice should be made and applied consistently. In figure 1, friends are declared first in the body of the class definition.

The ordering proposed here provides a uniform organization for the elements that comprise a C++ class definition. It should simplify a user's task in learning the interface of a class, and in locating information when making later references to this class.

---

reclassify the complex number class data as protected and add a derived class, rather than further modify the class (since new releases of it appear periodically, and it resides in a standard system library accessible by anyone).

## Extending C++ Stream I/O to Include Formats

Mark Rafter

...!mcvax!warwick!rafter

Computer Science Department

Warwick University

Coventry

England

### ABSTRACT

The `fmtio` library extends C++ stream I/O to include formatted I/O in the style of `stdio`. This extension is layered on top of stream I/O, and only requires minor changes to `<stream.h>`. The key traits of the original stream I/O system, namely extensibility and type-security, are retained. An example of its use is:

```
cout[ "log of %d is:%9f\n" ] << 5 << log(5);
```

which prints

```
log of 5 is: 1.609438
```

The `fmtio` library is presented as a suitable framework in which to conduct further experiments with formatted I/O systems. The methods used in the library are sketched, and its overall structure outlined. An example is given of how to equip a datatype with formatted I/O by interfacing it to the `fmtio` library.

KEY WORDS C++ C Formatted Input/Output Object-Oriented Streams Unix

## Introduction

The C++ stream I/O system is a great approach to I/O — it provides the programmer with a mechanism that is uniform, extensible and type-secure<sup>[1]</sup>. Stream I/O stops short of providing acceptable formatted I/O — its formatted output functions, **form**, **dec**, **hex** and **oct**, are clearly a stop-gap measure<sup>[2]</sup>. In particular, **form**, being a lightly disguised form of **sprintf**, delivers all of the problems that we associate with **sprintf** and its stdio cousins.

It was not clear to me whether formatted I/O had been omitted from the stream I/O library because of a mismatch between the formatted I/O idiom and the expressiveness of C++, or whether it was a routine piece of work that just hadn't got done. After some experimentation, I came to the conclusion that the answer lay somewhere between these two extremes.

Extensible, type-secure formatted I/O systems can be built in C++ — moreover the operator idiom of stream I/O, and the format-specifier idiom from stdio can both be retained. An example of such a library, **fmtio**, is described here. However, the methods used in its construction probably don't count as completely routine.

Is specifying features such as field-width, radix, justification and padding important to people, and if so why? The formatting facilities of stdio are breeding grounds for bugs, and this makes the real cost of using them quite high. The fact that people are willing to pay this cost might be taken as evidence that stdio's formatting facilities are valued. However, it is not the whole story.

The interface to stdio's formatting facilities can be made much safer. In C++ we could define overloaded **print** and **read** functions which would call **printf** or **scanf** in a manner appropriate to their argument types. This would be type-secure. It is even possible provide a safe interface to stdio in C, although then there would be an ugly proliferation of function names, **read\_int**, **read\_double**, **read\_complex** etc. .

Why do we not see safer interfaces to the stdio facilities more widely used? It is not just the formatting facilities of stdio that people value, it is the notational advantages of the formatted I/O idiom, in particular:

- The format-specifier notation is compact.
- The formatting information is gathered together in one place, and the data objects are gathered together in another.
- The format-specifier notation is simple and well-suited to routine tasks.

In effect, the formatted I/O idiom provides a little language<sup>[3]</sup> for describing I/O.

In developing **fmtio**, notational convenience was considered as important as both extensibility and type-security. Notational inconvenience was considered grounds for rejecting a design. Consequently, the simple to implement approach based on overloaded functions was rejected as too cumbersome.

Although the fmtio library achieves its aims — the demonstration of a convenient, extensible, type-secure formatted I/O library for C++ — it seems premature to recommend that fmtio be adopted as a standard. Something much better than stdio-like facilities should be possible — more experimentation and further work is called for. What the fmtio library does offer is a nice framework for building formatted I/O systems. This paper sketches the methods used in the fmtio library and outlines its overall structure. Only formatted output is discussed; formatted input uses the exactly the same mechanisms. The main ideas underlying fmtio are:

- The COOL assumption
- Control flow based on the COOL assumption
- Making C++ look COOL

Some of these ideas may have applications in other areas.

Throughout this paper the following program fragment will act as an example of the use of fmtio library:

```
cout["log of %d is:%9f\n"] << 5 << log(5);
```

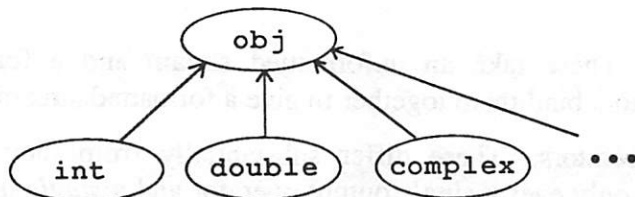
When executed, our example prints:

```
log of 5 is: 1.609438
```

The `%d` controls the format of the integer 5, and the `%9f` controls the format of the double `log(5)`.

### The COOL Assumption

The methods used in the construction of the fmtio library take their simplest form in a Completely Object-Oriented Language (COOL), i.e. one in which there is a type `obj` from which all other types are derived.



Of course, C++ is not such a language; but, in the interests of the explanation, we *temporarily* adopt the fiction that it is. In fact we go further than this and assume that the language has all the features that will make the implementation of fmtio easy. These are:

- All types are derived from the type `obj`.
- There is automatic coercion from any type to the base type `obj`.

- The base type **obj** is equipped with virtual **read** and **print** functions.

Each type has its own redefinition of **read** and **print** that handle the low-level details of formatted I/O for this type, and this type only. For any given datatype, the provision of these functions is the responsibility of the datatype designer.

Later we will show how to approximate this ideal situation in real C++. For now, we continue with our fiction and give the declaration of the **obj** datatype:

```
struct obj
{
    virtual void read( ostream & strm, char * fmt );
    virtual void print( istream & strm, char * fmt );
};
```

We illustrate the use of the virtual functions by reference to our example. Its execution should (somehow) result in the following two calls:

```
int(5).print( cout, "%d" );
```

```
double( log(5) ).print( cout, "%9f" );
```

The way in which these functions are provided with their arguments, and how they are called, is the subject of the next section.

## Control Flow Using the COOL Assumption

The **fmtio** library is almost entirely concerned with providing a control flow framework for the orderly execution of formatted I/O expressions. This framework is built with three main artifacts:

- The types **fostream** and **fistream**. These are the formatted stream types. Objects of these types contain information used in the execution of formatted I/O expressions.
- The index operators. These take an unformatted stream and a format-specifier (**char \***). The operators bind them together to give a formatted stream object.
- The formatted I/O operators. These differ substantially from their unformatted counterparts. There is only ever a *single* output operator and a *single* input operator. These are provided as part of the **fmtio** library. Both operators take a right-hand operand of type **obj** &. This, and the assumed automatic coercion of all types to the base type **obj**, allows the two formatted I/O operators to be applied to objects of any type.

The relationships between the above can be illustrated with our example:

then outputs the trailing plain text `"\n"`. At this point the formatted I/O expression is complete.

The operators of the `fmtio` library are entirely concerned with imposing control flow and providing controlled access to parts of the format-specifier. All of the details of the actual formatted I/O operations have been delegated to the virtual `read` and `print` functions associated with the various datatypes. Consequently the only formatted I/O modules that are linked into a program are the ones it uses. This improves on the `stdio` situation where all or none of the formatting modules are linked in.

Because the formatting information in the format-specifier is decoupled from the data objects to which it refers, it is possible for the two to be incompatible. This situation does not pose a type-security threat to `fmtio` in the same way that it does to `form`, `printf` and `scanf`. In `fmtio`, the formatting algorithm is chosen on the basis of the type of the object concerned, not on the basis of data in the format-specifier. The issue of incorrect data in a format-specifier should not be ignored; just as unformatted input operators should cope with incorrect data in an (external) data-stream, formatted I/O operators should *gracefully* handle incorrect data in an (internal) format-specifier.

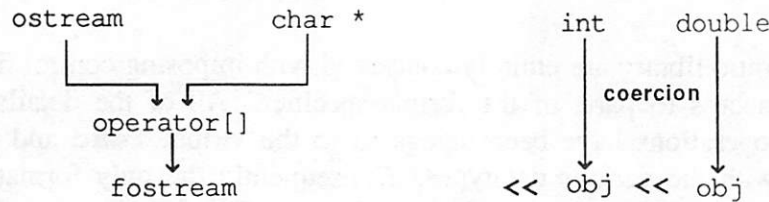
## Making C++ look COOL

To adapt the above scheme to C++ we must simulate a unified type hierarchy and its coercion mechanism. We don't attempt to do this in complete generality — that would be too hard — we need only simulate the unified type hierarchy for formatted I/O expressions. To do this:

- Take as the base of our hierarchy the type `obj`, declared in exactly the form shown above.
- The `obj` hierarchy will be a hierarchy of container classes. The objects in this hierarchy will refer to the right-hand operands in formatted I/O expressions. Each C++ type, `T`, is injected into the `obj` hierarchy by deriving a type `obj_T` from the type `obj`.
- All of this mechanism is hidden from the casual user of the `fmtio` library by equipping each of the container classes, `obj_T`, with a constructor taking an argument of type `T`. This constructor and the C++ type conversion rules will automatically inject an object of type `T` into the `obj` hierarchy when the object occurs as the right-hand operand of a formatted I/O operator.

This realisation of our formatted I/O expressions is a slight variant of our previous scheme:

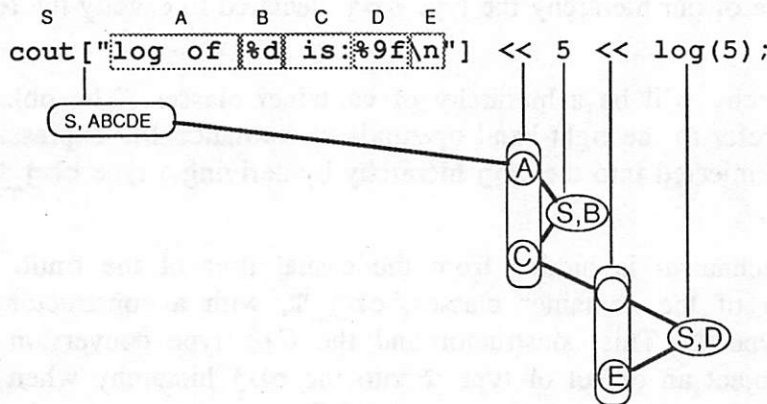
```
cout["log of %d is:%9f\n"] << 5 << log(5);
```



The execution of this formatted I/O expression starts when the index operator binds the unformatted stream **cout** and the format-specifier together creating a new formatted-stream object. Then control passes from left to right through each of the formatted I/O operators in turn, just as in the stream I/O system. As control progresses through the operators they deal with successive parts of the format-specifier. The operators record this progress in the formatted-stream object that was created by the index operator. The job of an I/O operator is in three parts:

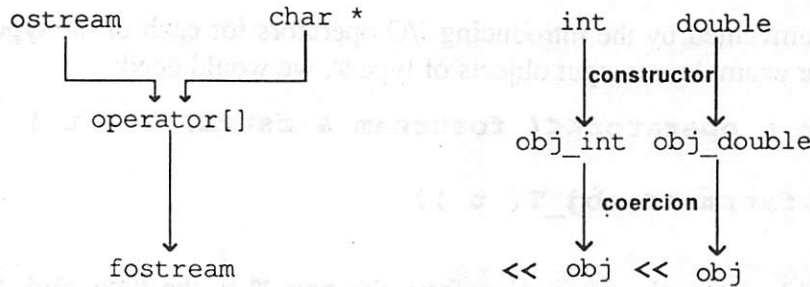
- Process any plain text in the format-specifier up to the next format control character.
- Make a copy of the format control part of the format-specifier. Call the virtual **read** or **print** function for the right-hand operand. Supply the unformatted stream to be used and the copy of part of format-specifier as the functions's arguments.
- Process any plain text remaining in the format-specifier.

We can trace the flow of control involved in executing our example. This shows which operators and functions deal with the various parts of the format specifier, and what they do with them.



Control flow starts with the index operator and passes to the left-hand output operator, which outputs the leading plain text "log of ". The left-hand output operator calls **int::print** with **cout** and "%d" as arguments. The function **int::print** outputs "5" on **cout** and returns to the left-hand output operator, which then outputs the trailing plain text " is:". Control flow then passes to the right-hand output operator, which has no leading plain text to deal with. The right-hand output operator then calls **double::print** with **cout** and "%9f" as arguments. The function **int::double** outputs " 1.609438" on **cout** and returns to the right-hand output operator, which

```
cout["log of %d is:%9f\n"] << 5 << log(5);
```



Although our scheme for COOL simulation in C++ may sound a little complicated, it is quite simple from the point of view of a datatype designer. To equip a datatype with formatted I/O, all that is required is a simple class declaration, in addition to the datatype-specific **read** and **print** functions. This class declaration would be unnecessary if type hierarchies could be suitably parameterised; alternatively the production of the class declaration could be automated with a suitable cpp macro — the author chooses not to do this.

As an example, here is the class declaration that is necessary to equip the type **complex** with formatted I/O:

```
struct  obj_complex : obj
{
    complex & datum;
    obj_complex( complex & d ): datum(d) {}
    virtual void  read( ostream & strm, char * fmt );
    virtual void print( istream & strm, char * fmt );
};
```

A quick and dirty version of **complex::read** and **complex::print** can be built with stream I/O or stdio library functions — this is not very interesting, and so is omitted here.

## Simplifications and Difficulties

The version of the **fmtio** library described here has the virtual **read** and **print** functions receiving their part of a format-specification via an argument that has type **char \***. The author has experimented with several variants of this mechanism, the most flexible of which is to make this argument an unformatted input stream<sup>[4]</sup>. The details of a format specification can then be read using unformatted, or even formatted, I/O operators. This makes handling field-widths etc. very easy.

The **fmtio** library depends very heavily on the details of the C++ type conversion rules. In particular, it is assumed that an object of type **obj\_T** will be constructed from an object of type **T** and that an **obj\_T** object will be automatically cast to its public base class **obj**. Unfortunately Release 1.2 of the C++ compiler from Bell Labs will not

implicitly apply both these conversions together.

The problem can be circumvented by the introducing I/O operators for each of the types in the `obj` hierarchy. For example to output objects of type `T`, we would need:

```
inline ostream & operator<<( ostream & ostrm, T & t )
{
    return ostrm << obj_T( t );
}
```

This operator just explicitly states the conversion from the type `T` to the type `obj_T`. The Release 1.2 compiler is then happy to apply the remaining conversion.

## Further Work

Layering the `fmtio` library on top of the stream I/O library is interesting, but has drawbacks. For example, it is easy to mix formatted and unformatted I/O operators in a single I/O expression but this doesn't work in a well. A version of the `fmtio` system that was integrated with the stream I/O library would solve this problem.

At the mundane level of the format-specifier syntax, something much better than the `stdio` style should be possible. For example, a syntax that defined, in a simple but flexible way, how much of the text following a '%' character should given to a `read` or `print` function would be a big help.

At a deeper level, we should probably examine more powerful ways of structuring streams of data; Rob Pike's use of structural regular expressions<sup>[5]</sup> looks like an interesting avenue.

## Summary

The development of the `fmtio` library has demonstrated that a convenient, extensible and type-secure formatted I/O library with the best features of both the `stdio` and stream I/O libraries can be built in C++. More important than the library itself are the methods used in its construction and its overall architecture — its framework. What is needed now is further experimentation to see what facilities we want to place within this framework.

## *References*

1. Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, New Jersey, 1986.
2. Bjarne Stroustrup, An Extensible I/O Facility for C++, pp57-70, USENIX Portland 1985 Summer Conference Proceedings.
3. Jon Bentley, Little Languages — Programming Pearls Column, Comm. Assoc. Comp. Mach, pp711-721, Oct 1986.
4. Mark Rafter, Formatted Streams: Extensible Formatted for C++ I/O Using Object-Oriented Programming, Research Report 107, Department of Computer Science, Warwick University, 1987.
5. Rob Pike, A Tutorial for the sam Command Language, Computing Science Technical Report No. 129, AT&T Bell Laboratories, Murray Hill, New Jersey, 1987.



# What is "Object-Oriented Programming"?

*Bjarne Stroustrup*

## ABSTRACT

"Object-Oriented Programming" and "Data Abstraction" have become very common terms. Unfortunately, few people agree on what they mean. I will offer informal definitions that appear to make sense in the context of languages like Ada, C++, Modula-2, Simula67, and Smalltalk. The general idea is to equate "support for data abstraction" with the ability to define and use new types and equate "support for object-oriented programming" with the ability to express type hierarchies. Features necessary to support these programming styles in a general purpose programming language will be discussed. The presentation centers around C++ but is not limited to facilities provided by that language.

## 1 Introduction

Not all programming languages can be "object oriented". Yet claims have been made to the effect that APL, Ada, Clu, C++, LOOPS, and Smalltalk are object-oriented programming languages. I have heard discussions of object-oriented design in C, Pascal, Modula-2, and CHILL. Could there somewhere be proponents of object-oriented Fortran and Cobol programming? I think there must be. "Object-oriented" has in many circles become a high-tech synonym for "good", and when you examine discussions in the trade press, you can find arguments that appear to boil down to syllogisms like:

Ada is good
Object oriented is good
-----
Ada is object oriented

This paper presents one view of what "object oriented" ought to mean in the context of a general purpose programming language.

§2 Distinguishes "object-oriented programming" and "data abstraction" from each other and from other styles of programming and presents the mechanisms that are essential for supporting the various styles of programming.

§3 Presents features needed to make data abstraction effective.

§4 Discusses facilities needed to support object-oriented programming.

§5 Presents some limits imposed on data abstraction and object-oriented programming by traditional hardware architectures and operating systems.

Examples will be presented in C++. The reason for this is partly to introduce C++ and partly because C++ is one of the few languages that supports both data abstraction and object-oriented programming in addition to traditional programming techniques. Issues of concurrency and of hardware support for specific higher-level language constructs are ignored in this paper.

## 2 Programming Paradigms

Object-oriented programming is a technique for programming a paradigm for writing "good" programs for a set of problems. If the term "object-oriented programming language" means anything it must mean a programming language that provides mechanisms that support the object-oriented style of programming well.

There is an important distinction here. A language is said to *support* a style of programming if it provides facilities that makes it convenient (reasonably easy, safe, and efficient) to use that style. A language does not support a technique if it takes exceptional effort or exceptional skill to write such programs; it merely *enables* the technique to be used. For example, you can write structured programs in Fortran, write type-secure programs in C, and use data abstraction in Modula-2, but it is unnecessarily hard to do because these languages do not support those techniques.

Support for a paradigm comes not only in the obvious form of language facilities that allow direct use of the paradigm, but also in the more subtle form of compile-time and/or run-time checks against unintentional deviation from the paradigm. Type checking is the most obvious example of this; ambiguity detection and run-time checks can be used to extend linguistic support for paradigms. Extra-linguistic facilities such as standard libraries and programming environments can also provide significant support for paradigms.

A language is not necessarily better than another because it possesses a feature the other does not. There are many example to the contrary. The important issue is not so much what features a language possesses but that the features it does possess are sufficient to support the desired programming styles in the desired application areas:

- [1] All features must be cleanly and elegantly integrated into the language.
- [2] It must be possible to use features in combination to achieve solutions that would otherwise have required extra separate features.
- [3] There should be as few spurious and "special purpose" features as possible.
- [4] A feature should be such that its implementation does not impose significant overheads on programs that do not require it.
- [5] A user need only know about the subset of the language explicitly used to write a program.

The last two principles can be summarized as "what you don't know won't hurt you." If there are any doubts about the usefulness of a feature it is better left out. It is *much* easier to add a feature to a language than to remove or modify one that has found its way into the compilers or the literature.

I will now present some programming styles and the key language mechanisms necessary for supporting them. The presentation of language features is not intended to be exhaustive.

### Procedural Programming

The original (and probably still the most commonly used) programming paradigm is:

*Decide which procedures you want;  
use the best algorithms you can find.*

The focus is on the design of the processing, the algorithm needed to perform the desired computation. Languages support this paradigm by facilities for passing arguments to functions and returning values from functions. The literature related to this way of thinking is filled with discussion of ways of passing arguments, ways of distinguishing different kinds of arguments, different kinds of functions (procedures, routines, macros, ...), etc. Fortran is the original procedural language; Algol60, Algol68, C, and Pascal are later inventions in the same tradition.

A typical example of "good style" is a square root function. It neatly produces a result given an argument. To do this, it performs a well understood mathematical computation:

```
double sqrt(double arg)
{
    // the code for calculating a square root
}

void some_function()
{
    double root2 = sqrt(2);
    // ...
}
```

>From a program organization point of view, functions are used to create order in a maze of algorithms.

## Data Hiding

Over the years, the emphasis in the design of programs has shifted away from the design of procedures towards the organization of data. Among other things, this reflects an increase in the program size. A set of related procedures with the data they manipulate is often called a *module*. The programming paradigm becomes:

*Decide which modules you want;  
partition the program so that data is hidden in modules.*

This paradigm is also known as the "data hiding principle". Where there is no grouping of procedures with related data the procedural programming style suffices. In particular, the techniques for designing "good procedures" are now applied for each procedure in a module. The most common example is a definition of a stack module. The main problems that have to be solved for a good solution are:

- [1] Provide a user interface for the stack (for example, functions `push()` and `pop()`).
- [2] Ensure that the representation of the stack (for example, a vector of elements) can only be accessed through this user interface.
- [3] Ensure that the stack is initialized before its first use.

Here is a plausible external interface for a stack module:

```
// declaration of the interface of module stack of characters
char pop();
void push(char);
const stack_size = 100;
```

Assuming that this interface is found in a file called `stack.h`, the "internals" can be defined like this:

```
#include "stack.h"
static char v[stack_size]; // "static" means local to this file/module
static char* p = v;        // the stack is initially empty

char pop()
{
    // check for underflow and pop
}

void push(char c)
{
    // check for overflow and push
}
```

It would be quite feasible to change the representation of this stack to a linked list. A user does not have access to the representation anyway (since `v` and `p` were declared static,

that is local to the file/module in which they were declared). Such a stack can be used like this:

```
#include "stack.h"

void some_function()
{
    char c = pop(push('c'));
    if (c != 'c') error("impossible");
}
```

Pascal (as originally defined) doesn't provide any satisfactory facilities for such grouping: the only mechanism for hiding a name from "the rest of the program" is to make it local to a procedure. This leads to strange procedure nestings and over-reliance on global data.

C fares somewhat better. As shown in the example above, you can define a "module" by grouping related function and data definitions together in a single source file. The programmer can then control which names are seen by the rest of the program (a name can be seen by the rest of the program *unless* it has been declared static). Consequently, in C you can achieve a degree of modularity. However, there is no generally accepted paradigm for using this facility and the technique of relying on static declarations is rather low level.

One of Pascal's successors, Modula-2, goes a bit further. It formalizes the concept of a module, making it a fundamental language construct with well defined module declarations, explicit control of the scopes of names (import/export), a module initialization mechanism, and a set of generally known and accepted styles of usage.

The differences between C and Modula-2 in this area can be summarized by saying that C only *enables* the decomposition of a program into modules, while Modula-2 *supports* that technique.

### Data Abstraction

Programming with modules leads to the centralization of all data of a type under the control of a type manager module. If one wanted two stacks, one would define a stack manager module with an interface like this:

```
class stack_id; // stack_id is a type
                // no details about stacks or stack_ids are known here

stack_id create_stack(int size); // make a stack and return its identifier
destroy_stack(stack_id);        // call when stack is no longer needed

void push(stack_id, char);
char pop(stack_id);
```

This is certainly a great improvement over the traditional unstructured mess, but "types" implemented this way are clearly very different from the built-in types in a language. Each type manager module must define a separate mechanism for creating "variables" of its type, there is no established norm for assigning object identifiers, a "variable" of such a type has no name known to the compiler or programming environment, nor do such "variables" do not obey the usual scope rules or argument passing rules.

A type created through a module mechanism is in most important aspects different from a built-in type and enjoys support inferior to the support provided for built-in types. For example:

```
void f()
{
    stack_id s1;
    stack_id s2;
```

```

s1 = create_stack(200);
// Oops: forgot to create s2

char c1 = pop(s1.push(s1,'a'));
if (c1 != 'c') error("impossible");

char c2 = pop(s2.push(s2,'a'));
if (c2 != 'c') error("impossible");

destroy(s2);
// Oops: forgot to destroy s1
}

```

In other words, the module concept that supports the data hiding paradigm enables this style of programming, but it does not support it.

Languages such as Ada, Clu, and C++ attack this problem by allowing a user to define types that behave in (nearly) the same way as built-in types. Such a type is often called an *abstract data type*<sup>†</sup>. The programming paradigm becomes:

*Decide which types you want;  
provide a full set of operations for each type.*

Where there is no need for more than one object of a type the data hiding programming style using modules suffices. Arithmetic types such as rational and complex numbers are common examples of user-defined types:

```

class complex {
    double re, im;
public:
    complex(double r, double i) { re=r; im=i; }
    complex(double r) { re=r; im=0; } // float->complex conversion

    friend complex operator+(complex, complex);
    friend complex operator-(complex, complex); // binary minus
    friend complex operator-(complex); // unary minus
    friend complex operator*(complex, complex);
    friend complex operator/(complex, complex);
    // ...
}

```

The declaration of class (that is, user-defined type) complex specifies the representation of a complex number and the set of operations on a complex number. The representation is *private*; that is, re and im are accessible only to the functions specified in the declaration of class complex. Such functions can be defined like this:

```

complex operator+(complex a1, complex a2)
{
    return complex(a1.re+a2.re,a1.im+a2.im);
}

```

and used like this:

<sup>†</sup> I prefer the term "user-defined type": "Those types are not "abstract"; they are as real as int and float."

Doug McIlroy. An alternative definition of *abstract data types* would require a mathematical "abstract" specification of all types (both built-in and user-defined). What is referred to as types in this paper would, given such a specification, be concrete specifications of such truly abstract entities.

```

complex a = 2.3;
complex b = 1/a;
complex c = a+b*complex(1,2,3);
// ...
c = -(a/b)+2;

```

Most, but not all, modules are better expressed as user defined types. For concepts where the "module representation" is desirable even when a proper facility for defining types is available, the programmer can declare a type and only a single object of that type. Alternatively, a language might provide a module concept in addition to and distinct from the class concept.

### Problems with Data Abstraction

An abstract data type defines a sort of black box. Once it has been defined, it does not really interact with the rest of the program. There is no way of adapting it to new uses except by modifying its definition. This can lead to severe inflexibility. Consider defining a type shape for use in a graphics system. Assume for the moment that the system has to support circles, triangles, and squares. Assume also that you have some classes:

```

class point{ /* ... */ };
class color{ /* ... */ };

```

You might define a shape like this:

```

enum kind { circle, triangle, square };

class shape {
    point center;
    color col;
    kind k;
    // representation of shape
public:
    point where() { return center; }
    void move(point to) { center = to; draw(); }
    void draw();
    void rotate(int);
    // more operations
};

```

The "type field" k is necessary to allow operations such as draw() and rotate() to determine what kind of shape they are dealing with (in a Pascal-like language, one might use a variant record with tag k). The function draw() might be defined like this:

```

void shape::draw()
{
    switch (k) {
        case circle:
            // draw a circle
            break;
        case triangle:
            // draw a triangle
            break;
        case square:
            // draw a square
            break;
    }
}

```

This is a mess. Functions such as draw() must "know about" all the kinds of shapes there are. Therefore the code for any such function grows each time a new shape is added to the system. If you define a new shape, every operation on a shape must be examined and (possibly) modified. You are not able to add a new shape to a system unless you have access to

the source code for every operation. Since adding a new shape involves "touching" the code of every important operation on shapes, it requires great skill and potentially introduces bugs into the code handling other (older) shapes. The choice of representation of particular shapes can get severely cramped by the requirement that (at least some of) their representation must fit into the typically fixed sized framework presented by the definition of the general type shape.

## Object-Oriented Programming

The problem is that there is no distinction between the general properties of any shape (a shape has a color, it can be drawn, etc.) and the properties of a specific shape (a circle is a shape that has a radius, is drawn by a circle-drawing function, etc.). Expressing this distinction and taking advantage of it defines object-oriented programming. A language with constructs that allows this distinction to be expressed and used supports object-oriented programming. Other languages don't.

The Simula67 inheritance mechanism provides a solution. First, specify a class that defines the general properties of all shapes:

```
class shape {
    point center;
    color col;
    // ...
public:
    point where() { return center; }
    void move(point to) { center = to; draw(); }
    virtual void draw();
    virtual void rotate(int);
    // ...
};
```

The functions for which the calling interface can be defined, but where the implementation cannot be defined except for a specific shape, have been marked "virtual" (the Simula and C++ term for "may be re-defined later in a class derived from this one"). Given this definition, we can write general functions manipulating shapes:

```
void rotate_all(shape* v, int size, int angle)
// rotate all members of vector "v" of size "size" "angle" degrees
{
    for (int i = 0; i < size; i++) v[i].rotate(angle);
}
```

To define a particular shape, we must say that it is a shape and specify its particular properties (including the virtual functions).

```
class circle : public shape {
    int radius;
public:
    void draw() { /* ... */ };
    void rotate(int) {} // yes, the null function
};
```

In C++, class circle is said to be *derived* from class shape, and class shape is said to be a *base* of class circle. An alternative terminology calls circle and shape subclass and superclass, respectively.

The programming paradigm is:

*Decide which classes you want;  
provide a full set of operations for each class;  
make commonality explicit by using inheritance.*

Where there is no such commonality data abstraction suffices. The amount of commonality between types that can be exploited by using inheritance and virtual functions is the litmus test of the applicability of object-oriented programming to an application area. In some areas, such as interactive graphics, there is clearly enormous scope for object-oriented programming. For other areas, such as classical arithmetic types and computations based on them, there appears to be hardly any scope for more than data abstraction and the facilities needed for the support of object-oriented programming seem unnecessary†.

Finding commonality among types in a system is not a trivial process. The amount of commonality to be exploited is affected by the way the system is designed. When designing a system, commonality must be actively sought, both by designing classes specifically as building blocks for other types, and by examining classes to see if they exhibit similarities that can be exploited in a common base class.

For attempts to explain what object-oriented programming is without recourse to specific programming language constructs see Nygaard<sup>13</sup> and Kerr<sup>9</sup>. For a case study in object-oriented programming see Cargill<sup>4</sup>.

### 3 Support for Data Abstraction

The basic support for programming with data abstraction consists of facilities for defining a set of operations for a type and for restricting the access to objects of the type to that set of operations. Once that is done, however, the programmer soon finds that language refinements are needed for convenient definition and use of the new types. Operator overloading is a good example of this.

#### Initialization and Cleanup

When the representation of a type is hidden some mechanism must be provided for a user to initialize variables of that type. A simple solution is to require a user to call some function to initialize a variable before using it. For example:

```
class vector {
    int sz;
    int* v;
public:
    void init(int size); // call init to initialize sz and v
                        // before the first use of a vector
    // ...
};

vector v;
// don't use v here
v.init(10);
// use v here
```

This is error prone and inelegant. A better solution is to allow the designer of a type to provide a distinguished function to do the initialization. Given such a function, allocation and initialization of a variable becomes a single operation (often called instantiation) instead of two separate operations. Such an initialization function is often called a constructor. In cases where construction of objects of a type is non-trivial, one often needs a complementary operation to clean up objects after their last use. In C++, such a cleanup function is called a destructor. Consider a vector type:

† However, more advanced mathematics may benefit from the use of inheritance: Fields are specializations of rings; vector spaces a special case of modules.

```

class vector {
    int sz;           // number of elements
    int* v;           // pointer to integers
public:
    vector(int);       // constructor
    ~vector();         // destructor
    int& operator[](int index); // subscript operator
};

```

The vector constructor can be defined to allocate space like this:

```

vector::vector(int s)
{
    if (s <= 0) error("bad vector size");
    sz = s;
    v = new int[s]; // allocate an array of "s" integers
}

```

The vector destructor frees the storage used:

```

vector::~vector()
{
    delete v; // deallocate the memory pointed to by v
}

```

C++ does not support garbage collection. This is compensated for, however, by enabling a type to maintain its own storage management without requiring intervention by a user. This is a common use for the constructor/destructor mechanism, but many uses of this mechanism are unrelated to storage management.

### Assignment and Initialization

Controlling construction and destruction of objects is sufficient for many types, but not for all. It can also be necessary to control all copy operations. Consider class vector:

```

vector v1(100);
vector v2 = v1; // make a new vector v2 initialized to v1
v1 = v2;        // assign v2 to v1

```

It must be possible to define the meaning of the initialization of v2 and the assignment to v1. Alternatively it should be possible to prohibit such copy operations; preferably both alternatives should be available. For example:

```

class vector {
    int* v;
    int sz;
public:
    // ...
    void operator=(vector&); // assignment
    vector(vector&);         // initialization
};

```

specifies that user-defined operations should be used to interpret vector assignment and initialization. Assignment might be defined like this:

```

vector::operator=(vector& a) // check size and copy elements
{
    if (sz != a.sz) error("bad vector size for =");
    for (int i = 0; i < sz; i++) v[i] = a.v[i];
}

```

Since the assignment operation relies on the "old value" of the vector being assigned to, the initialization operation *must* be different. For example:

```

vector::vector(vector& a) // initialize a vector from another vector
{
    sz = a.sz;           // same size
    v = new int[sz];      // allocate element array
    for (int i = 0; i < sz; i++) v[i] = a.v[i]; // copy elements
}

```

In C++, a constructor of the form `X(X&)` defines all initialization of objects of type `X` with another object of type `X`. In addition to explicit initialization constructors of the form `X(X&)` are used to handle arguments passed "by value" and function return values.

In C++ assignment of an object of class `X` can be prohibited by declaring assignment private:

```

class X {
    void operator=(X&); // only members of X can
    X(X&);              // copy an X
    ...
public:
    ...
};

```

Ada does not support constructors, destructors, overloading of assignment, or user-defined control of argument passing and function return. This severely limits the class of types that can be defined and forces the programmer back to "data hiding techniques"; that is, the user must design and use type manager modules rather than proper types.

### Parameterized Types

Why would you want to define a vector of integers anyway? A user typically needs a vector of elements of some type unknown to the writer of the vector type. Consequently the vector type ought to be expressed in such a way that it takes the element type as an argument:

```

class vector<class T> { // vector of elements of type T
    T* v;
    int sz;
public:
    vector(int s)
    {
        if (s <= 0) error("bad vector size");
        v = new T[sz = s]; // allocate an array of "s" "T"s
    }
    T& operator[](int i);
    int size() { return sz; }
    // ...
};

```

Vectors of specific types can now be defined and used:

```

vector<int> v1(100); // v1 is a vector of 100 integers
vector<complex> v2(200); // v2 is a vector of 200 complex numbers

v2[i] = complex(v1[x].v1[y]);

```

Ada, Clu, and ML support parameterized types. Unfortunately, C++ does not; the syntax used here is simply devised as an illustration. Where needed, parameterized classes are "faked" using macros. There need not be any run-time overheads compared with a class where all types involved are specified directly.

Typically a parameterized type will have to depend on at least some aspect of a type parameter. For example, some of the vector operations must assume that assignment is defined for objects of the parameter type. How can one ensure that? One solution to this problem is to require the designer of the parameterized class to state the dependency. For

example, "T must be a type for which = is defined". A better solution is not to or to take a specification of an argument type as a partial specification. A compiler can detect a "missing operation" if it is applied and give an error message such as. For example:

```
cannot define vector(non_copy)::operator[](non_copy&):
type non_copy does not have operator=
```

This technique allows the definition of types where the dependency on attributes of a parameter type is handled at the level of the individual operation of the type. For example, one might define a vector with a sort operation. The sort operation might use <, ==, and = on objects of the parameter type. It would still be possible to define vectors of a type for which '<' was not defined as long as the vector sorting operation was not actually invoked.

A problem with parameterized types is that each instantiation creates an independent type. For example, the type `vector<char>` is unrelated to the type `vector<complex>`. Ideally one would like to be able to express and utilize the commonality of types generated from the same parameterized type. For example, both `vector<char>` and `vector<complex>` have a `size()` function that is independent of the parameter type. It is possible, but not trivial, to deduce this from the definition of class `vector` and then allow `size()` to be applied to any vector. An interpreted language or a language supporting both parameterized types and inheritance has an advantage here.

## Exception Handling

As programs grow, and especially when libraries are used extensively, standards for handling errors (or more generally: "exceptional circumstances") become important. Ada, Algol68, and Clu each support a standard way of handling exceptions. Unfortunately, C++ does not. Where needed exceptions are "faked" using pointers to functions, "exception objects", "error states", and the C library signal and longjmp facilities. This is not satisfactory in general and fails even to provide a standard framework for error handling.

Consider again the vector example. What *ought* to be done when an out of range index value is passed to the subscript operator? The designer of the vector class should be able to provide a default behavior for this. For example:

```
class vector {
    ...
    except vector_range {
        // define an exception called vector_range
        // and specify default code for handling it
        error("global: vector range error");
        exit(99);
    }
}
```

Instead of calling an error function, `vector::operator[]()` can invoke the exception handling code, "raise the exception":

```
int& vector::operator[](int i)
{
    if (0 < i || sz <= i) raise vector_range;
    return v[i];
}
```

This will cause the call stack to be unraveled until an exception handler for `vector_range` is found; this handler will then be executed.

An exception handler may be defined for a specific block:

```

void f() {
    vector v(10);
    try {
        // errors here are handled by the local
        // exception handler defined below
        // ...
        int i = g(); // g might cause a range error using some vector
        v[i] = 7;    // potential range error
    }
    except {
        vector::vector_range:
        error("f(): vector range error");
        return;
    }
    // errors here are handled by the global
    // exception handler defined in vector

    int i = g(); // g might cause a range error using some vector
    v[i] = 7;    // potential range error
}

```

There are many ways of defining exceptions and the behavior of exception handlers. The facility sketched here resembles the ones found in Clu and Modula-2+. This style of exception handling can be implemented so that code is not executed unless an exception is raised† or portably across most C implementations by using `setjmp()` and `longjmp()`.††

Could exceptions, as defined above, be completely “faked” in a language such as C++? Unfortunately, no. The snag is that when an exception occurs, the run-time stack must be unraveled up to a point where a handler is defined. To do this properly in C++ involves invoking destructors defined in the scopes involved. This is not done by a C `longjmp()` and cannot in general be done by the user.

## Coercions

User-defined coercions, such as the one from floating point numbers to complex numbers implied by the constructor `complex(double)`, have proven unexpectedly useful in C++. Such coercions can be applied explicitly or the programmer can rely on the compiler to add them implicitly where necessary and unambiguous:

```

complex a = complex(1);
complex b = 1;          // implicit: 1 -> complex(1)
a = b+complex(2);
a = b+2;                // implicit: 2 -> complex(2)

```

Coercions were introduced into C++ because mixed mode arithmetic is the norm in languages for numerical work and because most user-defined types used for “calculation” (for example, matrices, character strings, and machine addresses) have natural mappings to and/or from other types.

One use of coercions has proven especially useful from a program organization point of view:

```

complex a = 2;
complex b = a+2; // interpreted as operator+(a,complex(2))
b = 2+a;         // interpreted as operator+(complex(2),a)

```

Only one function is needed to interpret “+” operations and the two operands are handled identically by the type system. Furthermore, class `complex` is written without any need to modify the concept of integers to enable the smooth and natural integration of the two concepts. This is in contrast to a “pure object-oriented system” where the operations would be interpreted like this:

† except possibly for some initialization code at the start of a program.

†† see the C library manual for your system.

```

a+2: // a.operator+(2)
2+a: // 2.operator+(a)

```

making it necessary to modify class integer to make 2+a legal. Modifying existing code should be avoided as far as possible when adding new facilities to a system. Typically, object-oriented programming offers superior facilities for adding to a system without modifying existing code. In this case, however, data abstraction facilities provide a better solution.

## Iterators

It has been claimed that a language supporting data abstraction must provide a way of defining control structures<sup>11</sup>. In particular, a mechanism that allows a user to define a loop over the elements of some type containing elements is often needed. This must be achieved without forcing a user to depend on details of the implementation of the user-defined type. Given a sufficiently powerful mechanism for defining new types and the ability to overload operators, this can be handled without a separate mechanism for defining control structures.

For a vector, defining an iterator is not necessary since an ordering is available to a user through the indices. I'll define one anyway to demonstrate the technique. There are several possible styles of iterators. My favorite relies on overloading the function application operator (`()`):

```

class vector_iterator {
    vector& v;
    int i;
public:
    vector_iterator(vector& r) { i = 0; v = r; }
    int operator() { return i < v.size() ? v.elem(i++) : 0; }
};

```

A `vector_iterator` can now be declared and used for a vector like this:

```

vector v(sz);
vector_iterator next(v);
int i;
while (i=next()) print(i);

```

More than one iterator can be active for a single object at one time, and a type may have several different iterator types defined for it so that different kinds of iteration may be performed. An iterator is a rather simple control structure. More general mechanisms can also be defined. For example, the C++ standard library provides a co-routine class<sup>15</sup>.

For many "container" types, such as vector, one can avoid introducing a separate iterator type by defining an iteration mechanism as part of the type itself. A vector might be defined to have a "current element":

```

class vector {
    int* v;
    int sz;
    int current;
public:
    // ...
    int next() { return (current++ < sz) ? v[current] : 0; }
    int prev() { return (0 <-- current) ? v[current] : 0; }
};

```

Then the iteration can be performed like this:

††† This style also relies on the existence of a distinguished value to represent "end of iteration". Often, in particular for C++ pointer types, 0 can be used.

```
vector v(sz);
int i;
while (i=v.next()) print(i);
```

This solution is not as general as the iterator solution, but avoids overhead in the important special case where only one kind of iteration is needed and where only one iteration at a time is needed for a vector. If necessary, a more general solution can be applied in addition to this simple one. Note that the "simple" solution requires more foresight from the designer of the container class than the iterator solution does. The iterator-type technique can also be used to define iterators that can be bound to several different container types thus providing a mechanism for iterating over different container types with a single iterator type.

### Implementation Issues

The support needed for data abstraction is primarily provided in the form of language features implemented by a compiler. However, parameterized types are best implemented with support from a linker with some knowledge of the language semantics, and exception handling requires support from the run-time environment. Both can be implemented to meet the strictest criteria for both compile time speed and efficiency without compromising generality or programmer convenience.

As the power to define types increases, programs to a larger degree depend on types from libraries (and not just those described in the language manual). This naturally puts greater demands on facilities to express what is inserted into or retrieved from a library, facilities for finding out what a library contains, facilities for determining what parts of a library are actually used by a program, etc.

For a compiled language facilities for calculating the minimal compilation necessary after a change become important. It is essential that the linker/loader is capable of bringing a program into memory for execution without also bringing in large amounts of related, but unused, code. In particular, a library/linker/loader system that brings the code for every operation on a type into core just because the programmer used one or two operations on the type is worse than useless.

## 4 Support for Object-Oriented programming

The basic support a programmer needs to write object-oriented programs consists of a class mechanism with inheritance and a mechanism that allows calls of member functions to depend on the actual type of an object (in cases where the actual type is unknown at compile time). The design of the member function calling mechanism is critical. In addition, facilities supporting data abstraction techniques (as described above) are important because the arguments for data abstraction and for its refinements to support elegant use of types are equally valid where support for object-oriented programming is available. The success of both techniques hinges on the design of types and on the ease, flexibility, and efficiency of such types. Object-oriented programming simply allows user-defined types to be far more flexible and general than the ones designed using only data abstraction techniques.

### Calling Mechanisms

The key language facility supporting object-oriented programming is the mechanism by which a member function is invoked for a given object. For example, given a pointer *p*, how is a call *p->f(arg)* handled? There is a range of choices.

In languages such as C++ and Simula67, where static type checking is extensively used, the type system can be employed to select between different calling mechanisms. In C++, two alternatives are available:

[1] A normal function call: the member function to be called is determined at compile time (through a lookup in the compiler's symbol tables) and called using the standard function call mechanism with an argument added to identify the object for which the function is called. Where the "standard function call" is not considered efficient enough, the programmer can declare a function inline and the compiler will attempt to inline expand its body. In this way, one can achieve the efficiency of a macro expansion without compromising the standard function semantics. This optimization is equally valuable as a support for data abstraction.

[2] A virtual function call: The function to be called depends on the type of the object for which it is called. This type cannot in general be determined until run time. Typically, the pointer *p* will be of some base class *B* and the object will be an object of some derived class *D* (as was the case with the base class *shape* and the derived class *circle* above). The call mechanism must look into the object and find some information placed there by the compiler to determine which function *f* is to be called. Once that function is found, say *D::f*, it can be called using the mechanism described above. The name *f* is at compile time converted into an index into a table of pointers to functions. This virtual call mechanism can be made essentially as efficient as the "normal function call" mechanism. In the standard C++ implementation, only five additional memory references are used.

In languages with weak static type checking a more elaborate mechanism must be employed. What is done in a language like Smalltalk is to store a list of the names of all member functions (methods) of a class so that they can be found at run time:

[3] A method invocation: First the appropriate table of method names is found by examining the object pointed to by *p*. In this table (or set of tables) the string "*f*" is looked up to see if the object has an *f()*. If an *f()* is found it is called; otherwise some error handling takes place. This lookup differs from the lookup done at compiler time in a statically checked language in that the method invocation uses a method table for the actual object.

A method invocation is inefficient compared with a virtual function call, but more flexible. Since static type checking of arguments typically cannot be done for a method invocation, the use of methods must be supported by dynamic type checking.

## Type Checking

The *shape* example showed the power of virtual functions. What, in addition to this, does a method invocation mechanism do for you? You can attempt to invoke *any* method for *any* object.

The ability to invoke any method for any object enables the designer of general purpose libraries to push the responsibility for handling types onto the user. Naturally this simplifies the design of libraries. For example:

```
class stack { // assume class any has a member next
    any* v;
    void push(any* p)
    {
        p->next = v;
        v = p;
    }
    any* pop()
    {
        if (v == 0) return error_obj;
        any* r = v;
        v = v->next;
        return r;
    }
};
```

It becomes the responsibility of the user to avoid type mismatches like this:

```
stack<any*> cs;

cs.push(new Saab900);
cs.push(new Saab37B);

plane* p = (plane*)cs.pop();
p->takeoff();

p = (plane*)cs.pop();
p->takeoff(); // Oops! Run time error: a Saab 900 is a car
              // a car does not have a takeoff method.
```

An attempt to use a car as a plane will be detected by the message handler and an appropriate error handler will be called. However, that is only a consolation when the user is also the programmer. The absence of static type checking makes it difficult to guarantee that errors of this class are not present in systems delivered to end-users. Naturally, a language designed with methods and without static types can express this example with fewer keystrokes.

Combinations of parameterized classes and the use of virtual functions can approach the flexibility, ease of design, and ease of use of libraries designed with method lookup without relaxing the static type checking or incurring measurable run time overheads (in time or space). For example:

```
stack<plane*> cs;

cs.push(new Saab900); // Compile time error:
                     // type mismatch: car* passed, plane* expected
cs.push(new Saab37B);

plane* p = cs.pop();
p->takeoff(); // fine: a Saab 37B is a plane

p = cs.pop();
p->takeoff();
```

The use of static type checking and virtual function calls leads to a somewhat different style of programming than does dynamic type checking and method invocation. For example, a Simula or C++ class specifies a fixed interface to a set of objects (of any derived class) whereas a Smalltalk class specifies an initial set of operations for objects (of any subclass). In other words, a Smalltalk class is a minimal specification and the user is free to try operations not specified whereas a C++ class is an exact specification and the user is guaranteed that only operations specified in the class declaration will be accepted by the compiler.

## Inheritance

Consider a language having some form of method lookup without having an inheritance mechanism. Could that language be said to support object-oriented programming? I think not. Clearly, you could do interesting things with the method table to adapt the objects' behavior to suit conditions. However, to avoid chaos, there must be some systematic way of associating methods and the data structures they assume for their object representation. To enable a user of an object to know what kind of behavior to expect, there would also have to be some standard way of expressing what is common to the different behaviors the object might adopt. This "systematic and standard way" would be an inheritance mechanism.

Consider a language having an inheritance mechanism without virtual functions or methods. Could that language be said to support object-oriented programming? I think not: the shape example does not have a good solution in such a language. However, such a language would be noticeably more powerful than a "plain" data abstraction language.

This contention is supported by the observation that many Simula67 and C++ programs are structured using class hierarchies without virtual functions. The ability to express commonality (factoring) is an extremely powerful tool. For example, the problems associated with the need to have a common representation of all shapes could be solved. No union would be needed. However, in the absence of virtual functions, the programmer would have to resort to the use of "type fields" to determine actual types of objects, so the problems with the lack of modularity of the code would remain†.

This implies that class derivation (subclassing) is an important programming tool in its own right. It can be used to support object-oriented programming, but it has wider uses. This is particularly true if one identifies the use of inheritance in object-oriented programming with the idea that a base class expresses a general concept of which all derived classes are specializations. This idea captures only part of the expressive power of inheritance, but it is strongly encouraged by languages where every member function is virtual (or a method). Given suitable controls of what is inherited (see Snyder<sup>17</sup> and Stroustrup<sup>18</sup>), class derivation can be a powerful tool for creating new types. Given a class, derivation can be used to add and/or subtract features. The relation of the resulting class to its base cannot always be completely described in terms of specialization; factoring may be a better term.

Derivation is another tool in the hands of a programmer and there is no foolproof way of predicting how it is going to be used and it is too early (even after 20 years of Simula) to tell which uses are simply mis-uses.

### Multiple Inheritance

When a class A is a base of class B, a B inherits the attributes of an A; that is, a B is an A in addition to whatever else it might be. Given this explanation it seems obvious that it might be useful to have a class B inherit from two base classes A1 and A2. This is called multiple inheritance<sup>22</sup>.

A fairly standard example of the use of multiple inheritance would be to provide two library classes displayed and task for representing objects under the control of a display manager and co-routines under the control of a scheduler, respectively. A programmer could then create classes such as

```
class my_displayed_task : public displayed, public task {  
    // my stuff  
};
```

```
class my_task : public task { // not displayed  
    // my stuff  
};
```

```
class my_displayed : public displayed { // not a task  
    // my stuff  
};
```

Using (only) single inheritance only two of these three choices would be open to the programmer. This leads to either code replication or loss of flexibility and typically both. In C++ this example can be handled as shown above with no significant overheads (in time or space) compared to single inheritance and without sacrificing static type checking<sup>19</sup>.

Ambiguities are handled at compile time:

† This is the problem with Simula67's inspect statement and the reason it does not have a counterpart in C++.

```

class A { public: f(); ... };
class B { public: f(); ... };
class C : public A, public B { ... };

void g() {
    C* p;
    p->f(); // error: ambiguous
}

```

In this, C++ differs from the object-oriented Lisp dialects that support multiple inheritance. In these Lisp dialects ambiguities are resolved by considering the order of declarations significant, by considering objects of the same name in different base classes identical, or by combining methods of the same name in base classes into a more complex method of the highest class.

In C++, one would typically resolve the ambiguity by adding a function:

```

class C : public A, public B {
public:
    f()
    {
        // C's own stuff
        A::f();
        B::f();
    }
    ...
}

```

In addition to this fairly straightforward concept of independent multiple inheritance there appears to be a need for a more general mechanism for expressing dependencies between classes in a multiple inheritance lattice. In C++, the requirement that a sub-object should be shared by all other sub-objects in a class object is expressed through the mechanism of a virtual base class:

```

class W { ... };

class Bwindow // window with border
: public virtual W
{ ... };

class Mwindow // window with menu
: public virtual W
{ ... };

class BMW // window with border and menu
: public Bwindow, public Mwindow
{ ... };

```

Here the (single) window sub-object is shared by the Bwindow and Mwindow sub-objects of a BMW. The Lisp dialects provide concepts of method combination to ease programming using such complicated class hierarchies. C++ does not.

## Encapsulation

Consider a class member (either a data member or a function member) that needs to be protected from "unauthorized access". What choices can be reasonable for delimiting the set of functions that may access that member? The "obvious" answer for a language supporting object-oriented programming is "all operations defined for this object"; that is, all member functions. A non-obvious implication of this answer is that there cannot be a complete and final list of all functions that may access the protected member since one can always add another by deriving a new class from the protected member's class and define a member function of that derived class. This approach combines a large degree of protection

from accident (since you do not easily define a new derived class "by accident") with the flexibility needed for "tool building" using class hierarchies (since you can "grant yourself access" to protected members by deriving a class).

Unfortunately, the "obvious" answer for a language oriented towards data abstraction is different: "list the functions that needs access in the class declaration". There is nothing special about these functions. In particular, they need not be member functions. A non-member function with access to private class members is called a friend in C++. Class complex above was defined using friend functions. It is sometimes important that a function may be specified as a friend in more than one class. Having the full list of members and friends available is a great advantage when you are trying to understand the behavior of a type and especially when you want to modify it.

Here is an example that demonstrate some of the range of choices for encapsulation in C++:

```
class B {
    // class members are default private
    int i1;
    void f1();
protected:
    int i2;
    void f2();
public:
    int i3;
    void f3();

    friend void g(B*); // any function can be designated as a friend
};
```

Private and protected members are not generally accessible:

```
void h(B* p)
{
    p->f1(); // error: B::f1 is private
    p->f2(); // error: B::f2 is protected
    p->f3(); // fine: B::f3 is public
}
```

Protected members, but not private members are accessible to members of a derived class:

```
class D : public B {
public:
    void g()
    {
        f1(); // error: B::f1 is private
        f2(); // fine: B::f2 is protected, but D is derived from B
        f3(); // fine: B::f3 is public
    }
};
```

Friend functions have access to private and protected members just like member functions:

```
void g(B* p)
{
    p->f1(); // fine: B::f1 is private, but g() is a friend of B
    p->f2(); // fine: B::f2 is protected, but g() is a friend of B
    p->f3(); // fine: B::f3 is public
}
```

Encapsulation issues increase dramatically in importance with the size of the program and with the number and geographical dispersion of its users. See Snyder<sup>17</sup> and Stroustrup<sup>18</sup> for more detailed discussions of language support for encapsulation.

## Implementation Issues

The support needed for object-oriented programming is primarily provided by the run-time system and by the programming environment. Part of the reason is that object-oriented programming builds on the language improvements already pushed to their limit to support for data abstraction so that relatively few additions are needed†.

The use of object-oriented programming blurs the distinction between a programming language and its environment further. Since more powerful special- and general-purpose user-defined types can be defined their use pervades user programs. This requires further development of both the run-time system, library facilities, debuggers, performance measuring, monitoring tools, etc. Ideally these are integrated into a unified programming environment. Smalltalk is the best example of this.

## 5 Limits to Perfection

A major problem with a language defined to exploit the techniques of data hiding, data abstraction, and object-oriented programming is that to claim to be a general purpose programming language it must

- [1] Run on traditional machines.
- [2] Coexist with traditional operating systems.
- [3] Compete with traditional programming languages in terms of run time efficiency.
- [4] Cope with every major application area.

This implies that facilities must be available for effective numerical work (floating point arithmetic without overheads that would make Fortran appear attractive), and that facilities must be available for access to memory in a way that allows device drivers to be written. It must also be possible to write calls that conform to the often rather strange standards required for traditional operating system interfaces. In addition, it should be possible to call functions written in other languages from a object-oriented programming language and for functions written in the object-oriented programming language to be called from a program written in another language.

Another implication is that an object-oriented programming language cannot completely rely on mechanisms that cannot be efficiently implemented on a traditional architecture and still expect to be used as a general purpose language. A very general implementation of method invocation can be a liability unless there are alternative ways of requesting a service.

Similarly, garbage collection can become a performance and portability bottleneck. Most object-oriented programming languages employ garbage collection to simplify the task of the programmer and to reduce the complexity of the language and its compiler. However, it ought to be possible to use garbage collection in non-critical areas while retaining control of storage use in areas where it matters. As an alternative, it is feasible to have a language without garbage collection and then provide sufficient expressive power to enable the design of types that maintain their own storage. C++ is an example of this.

Exception handling and concurrency features are other potential problem areas. Any feature that is best implemented with help from a linker is likely to become a portability problem.

The alternative to having "low level" features in a language is to handle major application areas using separate "low level" languages.

† This assumes that an object-oriented language does indeed support data abstraction. However, the support for data abstraction is often deficient in such languages. Conversely, languages that support data abstraction are typically deficient in their support of object-oriented programming.

## 6 Conclusions

Object-oriented programming is programming using inheritance. Data abstraction is programming using user-defined types. With few exceptions, object-oriented programming can and ought to be a superset of data abstraction. These techniques need proper support to be effective. Data abstraction primarily needs support in the form of language features and object-oriented programming needs further support from a programming environment. To be general purpose, a language supporting data abstraction or object-oriented programming must enable effective use of traditional hardware.

## 7 Acknowledgements

An earlier version of this paper was presented to the Association of Simula Users meeting in Stockholm. The discussions there caused many improvements both in style and contents. Brian Kernighan and Ravi Sethi made many constructive comments. Also thanks to all who helped shape C++.

## 8 References

- [1] Birtwistle, Graham et.al.: *SIMULA BEGIN*. Studentlitteratur, Lund, Sweden. 1971. Chartwell-Bratt Ltd, UK. 1980.
- [2] Bobrow, D. and Stefik, M.: *The LOOPS Manual*. Xerox Parc 1983.
- [3] Dahl, O.-J. and Hoare, C.A.R.: *Hierarchical Program Structures*. In *Structured Programming*. Academic Press 1972.
- [4] Cargill, Tom A.: *PI: A Case Study in Object-Oriented Programming*. SIGPLAN Notices, November 1986, pp 350-360.
- [5] C.C.I.T.T Study Group XI: *CHILL User's Manual*. CHILL Bulletin no 1. vol 4. March 1984.
- [6] Goldberg, A. and Robson, D.: *Smalltalk-80: The Language and its Implementation*. Addison-Wesley 1983.
- [7] Ichbiah, J.D. et.al.: *Rationale for the Design of the Ada Programming Language*. SIGPLAN Notices, June 1979.
- [8] Kernighan, B.W. and Ritchie, D.M.: *The C Programming Language*. Prentice-Hall 1978.
- [9] Kerr, Ron: *Object-Based Programming: A Foundation for Reliable Software*. Proceedings of the 14th SIMULA Users' Conference. August 1986, pp 159-165. An abbreviated version of this paper can be found under the title *A Materialistic View of the Software "Engineering" Analogy* in SIGPLAN Notices, March 1987, pp 123-125.
- [10] Liskov, Barbara et. al.: *Clu Reference Manual*. MIT/LCS/TR-225, October 1979.
- [11] Liskov, Barbara et. al.: *Abstraction Mechanisms in Clu*. CACM vol 20, no 8, August 1977, pp 564-576.
- [12] Milner, Robert: *A Proposal for Standard ML*. ACM Symposium on Lisp and Functional Programming. 1984, pp 184-197.

- [13]Nygaard, Kristen: *Basic Concepts in Object Oriented Programming*. SIGPLAN Notices, October 1986, pp 128-132.
- [14]Rovner, Paul: *Extending Modula-2 to Build Large, Integrated Systems*. IEEE Software, Vol. 3. No. 6. November 1986, pp 46-57.
- [15]Shapiro, Jonathan: *Extending the C++ Task System for Real-Time Applications*. Proc. USENIX C++ Workshop, Santa Fe, November 1987.
- [16]SIMULA Standards Group, 1984: *SIMULA Standard*. ASU Secretariat, Simula a.s. Post Box 150 Refstad, 0513 Oslo 5, Norway.
- [17]Snyder, Alan: *Encapsulation and Inheritance in Object-Oriented Programming Languages*. SIGPLAN Notices, November 1986, pp 38-45.
- [18]Stroustrup, Bjarne: *The C++ Programming Language*. Addison-Wesley, 1986.
- [19]Stroustrup, Bjarne: *Multiple Inheritance for C++*. Proceedings of the Spring'87 EUUG Conference. Helsinki, May 1987.
- [20]Stroustrup, Bjarne: *The Evolution of C++: 1985-1987*. Companion paper.
- [21]Stroustrup, Bjarne: *Possible Directions for C++: 1985-1987*. Companion paper.
- [22]Weinreb, D. and Moon, D.: *Lisp Machine Manual*. Symbolics, Inc. 1981.
- [23]Wirth, Niklaus: *Programming in modula-2*. Springer-Verlag, 1982.
- [24]Woodward, P.M. and Bond, S.G.: *Algol 68-R Users Guide*. Her Majesty's Stationery Office, London. 1974.

## An Object-Oriented Class Library for C++ Programs

Keith E. Gorlen

*National Institutes of Health, 9000 Rockville Pike, Bethesda, MD 20892, U.S.A.*

### SUMMARY

The Object-Oriented Program Support (OOPS) class library is a portable collection of classes similar to those of Smalltalk-80 that has been developed using the C++ programming language under the UNIX operating system. The OOPS library includes generally useful data types such as **String**, **Date**, and **Time**, and most of the Smalltalk-80 collection classes such as **OrderedCln** (indexed arrays), **LinkedList** (singly-linked lists), **Set** (hash tables), and **Dictionary** (associative arrays). Arbitrarily complex data structures comprised of OOPS and user-defined objects can be stored on disk files or moved between UNIX processes by means of an object I/O facility. Classes **Process**, **Scheduler**, **Semaphore**, and **SharedQueue** provide multiprogramming with co-routines. This paper gives a brief introduction to object-oriented programming and how it is supported by the C++ programming language. An overview of the OOPS library is also presented, followed by a programming example. The implementation details of two of the class library's more interesting features, object I/O and processes, are described. The paper concludes with a discussion of the differences between the OOPS library and Smalltalk-80 and some observations based on our programming experience with C++ and OOPS.

### KEY WORDS

Object-oriented programming  
Smalltalk-80 Classes  
inheritance

C++ programming language  
encapsulation  
dynamic binding

## INTRODUCTION

Contemporary user interfaces are greatly influenced by Smalltalk-80,<sup>1-4</sup> a programming system developed at the Xerox Palo Alto Research Center. The Smalltalk-80 user interface is a classic example of a "direct-manipulation" user interface, one in which a user controls a graphical model of a system by using a pointing device, such as a mouse, instead of typing commands on a keyboard. For example, a user of a direct-manipulation interface to a file system might delete a file by moving a picture of a file folder onto a picture of a trash can with the mouse instead of typing the command "del myfile" on the keyboard. Direct-manipulation user interfaces are easier to learn and remember than conventional user interfaces, making them ideal for the casual user. However, they are also much more difficult to program.

Smalltalk-80 supports object-oriented programming, a particularly good technology for implementing direct-manipulation user interfaces, but in a form not suitable for many applications. Smalltalk-80 is dynamic and must be interpreted rather than compiled for most commercially available 32-bit workstation computer systems. It therefore lacks the efficiency required for time-critical use. Smalltalk-80 also suffers from isolation. It is itself a complete system that is not widely available under popular operating systems such as UNIX, and it is not compatible with libraries written in other languages, so existing libraries for input/output, graphics, communications, database management, and numeric computation are inaccessible.

An attractive solution to the problems of inefficiency and isolation is to incorporate support for object-oriented programming into C, the language of choice under UNIX and many other operating systems. The C++ programming language<sup>5</sup> is a successor to C that corrects most of the deficiencies of C and adds many powerful new features, including support for object-oriented programming, while remaining highly compatible with C. Although a C++ programmer can use C libraries, the lack of a library of common classes prevents users from taking full advantage of the object-oriented features of C++. The Object-Oriented Program Support (OOPS) class library was developed to make some of the functionality of Smalltalk-80 available to programs written in C++ and run in a UNIX environment.

Readers familiar with C++ may wish to read the example program in Appendix A to get an idea of how to use the OOPS class library and what it can do. This example illustrates several features of object-oriented programming in general and of the OOPS class library in particular:

- A great deal of programming effort is saved by using existing OOPS library classes: `Dictionary`, which uses hash tables to implement associative arrays, `SortedCltn`, which uses arrays to implement lists of objects that can be ordered, `String`, and `Date`.
- The classes are general – the hash table algorithm works with any kind of object as a key, for example. This makes software more reusable.
- The class library is flexible because classes can be customized for a particular application. In this example, a specialized class `DateDict`, which deals specifically with dates and sorted collections of strings, is made from the general class `Dictionary`, which can handle any type of object. As shown below, users can also extend the library by adding their own classes.

- It is easy to construct and manipulate complex data structures.
- Improvements in productivity are realized. After errors noted by the compiler were fixed, this program ran correctly the first time.

## OBJECT-ORIENTED PROGRAMMING CONCEPTS

Each OOPS *class* specifies an abstract data type. A user program, or *client*, can create one or more *instances* of a class; such an instance is also called an *object*. An object consists of some data, or *member variables* and a set of operations, or *member functions*, that can be applied to the object's data. Both the member variables and functions are defined by the object's class.

Normally, a program cannot directly read or modify the member variables of an object; it can only call the object's member functions. This is called *encapsulation* and it has several advantages:

- The interface between the client and the object is well-defined and localized in the object's class definition. The implementation is hidden from the client, thus allowing it to change with the assurance that the client will still function properly. For example, a class implementing a stack data structure could initially be implemented with an array, and later it could be changed to use a linked list. As long as the calling sequences and semantics of the member functions are not changed, client programs will work because encapsulation prevents them from relying on a particular implementation of the stack abstract data type.
- A well-designed implementation also hides the complexity of its operation from the client, making objects easier to use.
- Debugging is easier when an object's member variables are found to have improper values. With few exceptions, the problem is localized to the object's member functions since only these have access to the member variables.

Another key concept of object-oriented programming is *inheritance*, a mechanism by which the member variables and functions of one class, called the *base* class or *superclass*, are included in another class, called the *derived* class or *subclass*. A base class may have multiple derived classes, and a derived class may in turn serve as the base class for other derived classes, producing a tree-structured organization of classes such as shown in Fig. 1.

A derived class distinguishes itself from its base class by adding member variables, adding member functions, or redefining inherited member functions. In the last case, a member function declared in a base class may have several definitions since it may be redefined in multiple derived classes. When the function is called to perform an operation on an object, the definition actually used is determined at execution time based on the class of the object. This is called *dynamic binding*. Dynamic binding encourages placing the code that deals with a particular class of object in the implementation of the object's class, rather than in the client program, thereby making the client program more general.

For example, Fig. 1 shows how classes for drawing simple geometric figures on a display might be organized. Note that the most general class, *Object*, is the root of the tree, and that the classes become more specialized the further they are from the root.

Class Shape contains no member variables and declares member functions that are applicable to any shape, such as `draw()`, `move()`, and `rotate()`. Since these functions cannot be implemented usefully in class Shape, they are defined to print an error message if called.

Class Polygon is derived from class Shape, and contains member variables that represent a polygon as an array of the coordinates of the vertices (`Point* vertex`) and their number (`int nvertex`). Class Polygon reimplements the functions `draw()`, `move()`, and `rotate()` to operate on the array of vertices, which is actually an array of instances of class Point.

Classes Quadrilateral and Triangle are derived from class Polygon and simply initialize the inherited array of vertices with an appropriate number of coordinates. They inherit suitable implementations of `draw()`, `move()`, and `rotate()` from class Polygon.

Class Circle is derived directly from class Shape because it requires different member variables for its representation than a polygon. A circle is described by the coordinates of its center (`Point center`), and its radius (`int radius`). Note that the member variable `center` is actually an instance of class Point, illustrating how a class can use instances of other classes as member variables. Class Circle reimplements the functions `draw()`, `move()`, and `rotate()` to operate on the center and radius.

Now code can be written to operate on instances of class Shape by applying the member functions `draw()`, `move()`, and `rotate()`. At run time, these instances will actually be instances of class Quadrilateral, Triangle, or Circle. The dynamic binding mechanism insures that the implementation of the function appropriate to the particular shape is called. For example, when `draw()` is called for an instance of class Triangle, the definition of `draw()` in class Polygon is executed; but when `draw()` is called for an instance of class Circle, the definition of `draw()` in class Circle is executed.

Inheritance and dynamic binding also make programs easier to extend. For example, a new geometric figure such as a circular arc can be added simply by using Circle as the base class for a new class, Arc, which defines additional member variables to hold the starting and stopping angles and reimplements the member functions `draw()` and `rotate()`. Class Arc inherits the member variables `center` and `radius` and the member function `move()` from class Circle. Dynamic binding lets existing code that manipulates instances of class Shape work automatically for instances of class Arc without having to be recompiled.

Similarly, the OOPS class library is easily extended by using the OOPS class Object as the base for user-defined classes. Dynamic binding enables the OOPS classes that manipulate instances of class Object to work automatically for instances of user-defined classes if the user reimplements some inherited member functions in his classes according to a few design rules.

## C++ SUPPORT FOR OBJECT-ORIENTED PROGRAMMING

C++ supports encapsulation and inheritance by means of the class declaration. For example:

```

class Point : public Object {
    int xc,yc;                // x-y coordinates of the point
public:
    int x() { return xc; }    // read x coordinate
    int y() { return yc; }    // read y coordinate
    Point(int px, int py) { xc=px; yc=py; } // constructor
    ~Point();                 // destructor
    virtual void printOn(ostream&);        // print on output stream
};

```

declares the class `Point` with private member variables `xc` and `yc` and public member functions `x()`, `y()`, `Point()`, `~Point()` and `printOn()`. Since `xc` and `yc` are private, they can be accessed only by the member functions. In this example, a client program may read the values of `xc` and `yc` by calling the member functions `x()` and `y()`, but may not write into these variables.

The member variables of a class are referenced using the `.` or `->` operators in the same way that structure members are referenced in C:

```

Point a;           // declare an object of class Point
Point* p;          // declare a pointer to an object of class Point
a.xc               // reference to member variable xc of object a
p->xc              // reference to member variable xc of object
                  // pointed to by p

```

Member functions are called in an analogous manner:

```

a.x()              // call of member function x() on object a
p->x()              // call of member function x() on object
                  // pointed to by p

```

Within the body of a member function the reserved name `this` can be used to refer to the particular instance for which the member function was called. Since member functions frequently reference member variables and other member functions, this is implied when any member variable or function is referenced without an explicit object. Thus, the member function `x()` is written as `return xc`, which is interpreted as `return this->xc`, and this is assigned the address of `a` when a call such as `a.x()` is made.

A member function with the same name as its class, such as `Point()`, is a special function called a *constructor*. Constructor functions create a new instance of their class and initialize it, and are implicitly called whenever an object of their class is declared or is allocated via the C++ `new` operator. Similarly, a member function with the same name as its class prefixed by the character `~`, such as `~Point()`, is a special function called a *destructor*, which can be used to "clean up" when an object is deleted. Destructor functions are called automatically when an object that is declared as a local variable of a block goes out of scope or when an object is de-allocated by the C++ `delete` operator.

Constructor and destructor functions are also invoked for statically allocated objects, in which case they are known as *static constructors* and *static destructors*. Static constructors are called before the `main()` program is executed, and static destructors are called after the `main()` program finishes or `exit()` is called. They are invaluable for

initializing/finalizing objects used by libraries; for example, the C++ stream I/O library statically declares `cout` as an `ostream` (output stream) object. The `ostream` constructor allocates buffer space from the free store and connects `cout` to the standard output file, and the `ostream` destructor flushes the buffer and deallocates it. Thus, a client program may use `cout` without explicitly calling library routines to open or close it.

The keyword `virtual` specifies that dynamic binding is to be used for the function to which it is applied, as in the declaration of the function `printOn()` in the above example.

The class name in the class declaration may be followed by a specification of a base class (e.g. `public Object`), denoting that the members of the base class are inherited by the declared derived class. Private members of the base class are inherited, but cannot be accessed by the derived class, thus preserving their encapsulation. Public members of the base class are inherited as private members of the derived class by default, but usually they are caused to be inherited as public members by qualifying the name of the base class with the keyword `public`, as in the above example.

In a class declaration, the declaration of a member function may include the code to implement it enclosed in braces, as in the functions `x()`, `y()`, and `Point()`. This signifies that the function is to be compiled *inline* for efficiency, much like a macro expansion. Inline functions make it practical to define trivial operations on objects as member functions, thus encouraging the use of private member variables.

C++ allows several classes to have member functions and variables with the same names, and provides the scope resolution operator `::` for situations when it is necessary to explicitly specify the member function. Thus, `Object::printOn()` refers to the function `printOn()` that is a member of class `Object`, while `Point::printOn()` refers to the function `printOn()` that is a member of class `Point`.

Several functions may also have the same name if they can be distinguished by the number or types of their arguments; such functions are said to be *overloaded*. For example, the functions `Point::x()` and `Point::y()` can be overloaded to allow clients of class `Point` to modify the values of the member variables `xc` and `yc`:

```
class Point : public Object {
    int xc,yc;                // x-y coordinates of the point
public:
    int x() { return xc; }    // read x coordinate
    int y() { return yc; }    // read y coordinate
    int x(int newx) { return xc = newx; } // write x coordinate
    int y(int newy) { return yc = newy; } // write y coordinate
    // ...
};
```

C++ operators may be defined for class objects by overloading them; for example, addition of `Point` objects may be defined by adding the following member function definition to class `Point`:

```
class Point : public Object {
    int xc,yc;                // x-y coordinates of the point
    // ...
```

```
Point operator+(Point p) { return Point(xc+p.xc,yc+p.yc); }
```

Client programs can then apply the operator "+" to objects of class Point:

```
Point a,b,c;
```

```
...
```

```
c = a+b;
```

## OVERVIEW OF OOPS CLASSES

Class Object is the most general OOPS class. All other OOPS classes, and classes written by a user to extend OOPS, are ultimately derived from Object and inherit member functions that enable objects to be compared, copied, printed, read, stored, and retrieved. They also allow the class of an object to be tested and provide error handling.

As an example, consider the simple OOPS library class Point, which is derived directly from class Object. If an instance of class Point named p is declared, then the following are a few of the member functions that can be called for p:

- |                     |   |
|---------------------|---|
| p.className()       | returns a pointer to a character string containing the name of the class which p is an instance of, in this case "Point".   |
| p.isKindOf(aClass)  | tests to see if p is an instance of the specified class, or of a class derived from the specified class. Thus, p.isKindOf(class_Object) is true, and p.isKindOf(class_String) is false. The argument to isKindOf() is a global instance of class Class, which is described below.   |
| p.printOn(cout)     | causes p to be printed on the output stream cout.   |
| p.isEqual(anObject) | tests to see if p is equal to the specified object. This involves two tests. First, anObject must be checked to make sure that it is also an instance of class Point, or of a class derived from class Point, so that the equality test is meaningful. Then, if this is true, the objects are tested for equality in a manner determined by the particular class. |
| p.isSame(anObject)  | tests to see if p and the specified object are the <i>same</i> object; i.e., are stored at the same location in memory.   |

Some of these functions, such as className() or isKindOf() are completely implemented by class Object and can be applied to objects of any class, including those added by the user, without the need for additional code. Other functions, such as printOn() or isEqual() are virtual functions that must be redefined by a derived class to be useful. Class Object implements these functions to issue an error message if they are applied to objects of a class that does not provide its own implementation.

An important virtual function declared in class Object is the function isA(), which

returns a pointer to an instance of class `Class` that describes the object. Instances of `Class` contain information such as the name of the class, the size of instances of the class, and a pointer to the instance of `Class` that describes the base class of the class. Functions like `className()` and `isKindOf()` work by calling the `isA()` function and using the information found in the class descriptor.

There is one instance of `Class` for each OOPS class linked with a program. The implementation of each OOPS class declares a static instance of `Class`, which is initialized by the static constructor for `Class`. This constructor links all the instances together into a tree structure that reflects the hierarchical organization of OOPS classes. The constructor also builds a data structure called the `classDictionary`, which allows a class descriptor to be found given the class's name.

`Class` is derived from class `Object`, so instances of `Class` can be manipulated by the member functions inherited from class `Object`. In particular, the function `isA()` may be applied to an instance of `Class`, and it will return a pointer to the instance of `Class` that describes instances of `Class`. This is also known as the *metaclass* object. The metaclass object exists mainly to eliminate some special cases in the OOPS code, and is of little concern to users of OOPS.

Fig. 2 shows the OOPS data structure for an instance of class `Point`. Each object inherits a pointer, the `vp`tr, to a table of pointers to the virtual functions defined for its class, the `vtbl`. The `vtbl` also contains pointers to all the inherited virtual functions. Both the `vp`tr and `vtbl` are completely managed by the C++ translator and are not accessible to the user. When a virtual function is called, C++ generates code to call the function via the pointer found at some constant offset reserved for that function in the `vtbl`. This enables objects of different classes to have different implementations for the same function, and is the mechanism that implements dynamic binding.

A good example of how inheritance and dynamic binding make code more general and reusable is provided by the OOPS library classes `Collection`, `Set`, `Dictionary`, `IdentDict`, and `Assoc`. The organization of these classes in the OOPS class hierarchy is shown in Fig. 3.

Class `Collection` is the base class for classes known as *container* classes, so called because they contain collections of pointers to objects. The functions that are applicable to all container classes are declared in `Collection`. These include `add()`, which adds an object to a collection, `remove()`, which removes an object from a collection, and `includes()`, which tests if an object belongs to a collection. Class `Collection` is an example of an *abstract class* – a class that is intended to be used exclusively as a base class. A client cannot create instances of class `Collection` because the constructor function for this class is not public.

One of the classes derived from `Collection` is class `Set`, which manages the object pointers it contains by using a hash table with open addressing. Any object belonging to a class which implements the virtual functions `hash()` and `isEqual()` can be added to a `Set`. The implementations of `add()`, `remove()`, and `includes()` for class `Set` call the virtual function `Set::findIndexOf()`, which uses `hash()` and `isEqual()` to locate objects in the hash table. Class `Set` also overloads the "&", "|", and "-" operators so they form the intersection, union, or difference of two sets.

Class Dictionary, which is derived from class Set, implements associative arrays by redefining the virtual function `add()` so that only associations (i.e., instances of class Assoc) can be added to the hash table inherited from class Set. An instance of class Assoc contains a pointer to a *key* object and a pointer to a *value* object, and implements `hash()` and `isEqual()` to call the corresponding functions on the key object. As a result, when the function `Set::findIndexOf()` is applied to a Dictionary it will find an association with a matching key object, if one exists, from which the pointer to the associated value object can be retrieved. Class Dictionary defines the virtual functions `assocAt()` and `atKey()` to retrieve the association and value object pointers, respectively, that match a given key.

Class Dictionary is further specialized by its derived class `IdentDict`, which can be used to associate a value object with a particular instance of any class. This is accomplished simply by reimplementing the virtual function `findIndexOf()` in class `IdentDict` so that the hash function is computed by using the *address* of the key object rather than by using `hash()` on the key object, and by comparing keys with `isSame()` rather than `isEqual()`. The functions `assocAt()` and `atKey()`, inherited from class Dictionary, retrieve the association and value object pointers associated with the object at a particular address when applied to an `IdentDict`.

These are just a few of the classes currently available in the OOPS class library. Table I contains a complete list.

## AN EXAMPLE APPLICATION OF OOPS

Figs. 4 and 5 show how to organize a program to manage forms on a terminal screen as a demonstration of how OOPS library classes can be combined with application-specific classes to manage a complex data structure. The OOPS library is extended with user-written classes, but these classes utilize instances of OOPS library classes to do most of the work.

Class Form consists of an `OrderedCltn` of the individual fields in the form, represented by class Field. The order in which the fields are normally filled in corresponds to their order in the `OrderedCltn`, so that a field's predecessor and successor can be easily found. Class Form also has a Dictionary that associates field names, represented by instances of class String, with the fields so that they can be accessed independently of their position in the form.

Since all form fields have a name and occupy a rectangular area on the screen, class Field contains a member variable of type `String*` to point to an instance of class String with the field's name and a member variable of class Rectangle to describe the field's screen position. Of course, a form may have many different types of fields, both general-purpose and application-specific: numeric, decimal, dollar, alphabetic, part number, etc. These are represented by various classes derived from class Field: `NumberField`, `DecimalField`, `DollarField`, and `AlphaField`.

Classes `OrderedCltn`, `Dictionary`, `String`, `Rectangle`, and `Assoc` are part of the OOPS class library; the other classes are application-specific and are supplied by the user.

`OrderedCltn`, which is derived from `Collection`, contains objects that are ordered by the sequence in which they are added. The objects in an `OrderedCltn` can be accessed

by an integer index, much like an array. However, an `OrderedCltn` can hold instances of a mixture of classes, whereas all of the elements in an array must be of the same type. In this example, the member variable `fields` of class `Form` holds pointers to various derived classes of `Field` in the order in which they are to be filled in on the screen. Class `Form` can use the variable `fields` to implement the member functions like `fillIn()`, which fills in the various types of fields in order by calling `Field::fillIn()` for each field:

```
virtual void Form::fillIn()
{
    for (register i=0; i<fields.size(); i++) ((Field*)&fields[i])->fillIn();
}
```

The expression `(Field*)&fields[i]` accesses object `i` in the `OrderedCltn` `fields`, and converts it to type `Field*` so that the virtual function `Field::fillIn()` can be called. This conversion, or *cast*, is necessary because `fields[i]` is of type `Object&`, and `fillIn()` is not defined for class `Object`. Casts must always be used cautiously. In this case, we know that the cast is safe because the function `Form::addField(Field& f)` (described below) is the only function a client can use to add an object to `fields`, and `addField()` only accepts objects of class `Field` as arguments.

Class `String` provides a convenient string manipulation package. Functions include a full complement of comparison operators, conversion to upper case, lower case, and ASCII, subscripting, printing, storing, and concatenation. The operator `(position,length)` allows extraction of and assignment to substrings. Conversion of `String` objects to and from `const char*` is defined, permitting `Strings` to be used as arguments to C library routines such as `open()`, for example.

Class `Rectangle` represents rectangular regions with sides parallel to the x-y axis as two instances of class `Point`. Functions are provided for calculating area, width, height, and the intersection or union of two rectangles. A point or rectangle can be tested to see if it is contained by a rectangle, or if two rectangles intersect. As in `Smalltalk-80`, no functions for actually drawing a rectangle on a display are provided. These are presumably added by a user-defined graphics class.

Fig. 5 shows the data structure for an instance of class `Form` that consists of two fields, a `NumberField` and an `AlphaField`. The application-specific code that constructs this data structure illustrates how the OOPS library classes are used. The constructor for class `Field` initializes the field name, which is a pointer to an instance of class `String`, and the field location, which is an instance of class `Rectangle`:

```
Field::Field(char* fname, Rectangle& floc)
{
    namePtr = new String(fname); // create String for field name
    location = floc;
}
```

Class `Field` also defines the function `name()` so that the name of a field can be determined:

```
const String& Field::name() { return *namePtr; }
```

The constructors for classes derived from `Field` use this class's constructor when creating instances of their own class. For example, here are the constructors for classes `NumberField` and `AlphaField`:

```
NumberField::NumberField(char* fname, Rectangle& floc, int fval)
    : (fname,floc)           // parameters for base class constructor
{
    value = fval;
}
```

```
AlphaField::AlphaField(char* fname, Rectangle& floc, char* fval)
    : (fname,floc)           // parameters for base class constructor
{
    value = fval;
}
```

The parameters for the constructor `Field::Field` are supplied on the second line of each function definition following the ":". The only difference between these constructors is the type of value they hold.

An instance of class `Field` can be added to a `Form` object by the function `Form::addField()`. This function adds the field to the list of fields in the form, fields, creates an association with the field's name as the key and the field object as the value, and then adds this association to the Dictionary `fieldNames`:

```
void Form::addField(Field& f)
{
    fields.add(f);
    fieldNames.add(*new Assoc(f.name(),f));
}
```

Creating the data structure shown in Fig. 5 requires declaring an instance of an empty form, named `aForm` in this example, and adding instances of the two fields to it by calling `Form::addField()`:

```
Form aForm;           // create an empty form
aForm.addField(
    *new NumberField(
        "Part Number",
        Rectangle(Point(10,20),Point(50,30)),
        123654));
aForm.addField(
    *new AlphaField(
        "Description",
        Rectangle(Point(60,20),Point(100,30)),
        "large blue widget"));
```

As illustrated by this example, only a small amount of code is needed to combine a variety of classes from the OOPS library into new, application-specific classes that manipulate a fairly complex data structure. Most of the real work such as storage management, searching, string comparison, etc. is done by the OOPS classes.

## OBJECT INPUT/OUTPUT

The OOPS Object I/O Facility enables arbitrarily complex data structures composed of OOPS and user-defined objects to be saved on disk files or moved between UNIX® processes. If, for example, a program has created an instance of a `Form` named `aForm`, such as shown in Fig.5, and opened an output file on an `ostream` named `out`, it can be written to the file by executing the statement `aForm.storeOn(out)`. It can then be read in at a later time with the statement `readFrom(in,"Form",aForm)`.

Class `Object` implements the function `storeOn(ostream&)` as a virtual function. When `storeOn()` is called to store the first object in a data structure, it constructs an instance of an `IdentDict` named `storeOnDict` to keep track of which objects have been stored as the data structure is traversed so that multiple references to the same object can be resolved. Next, `storeOn()` checks the `storeOnDict` to see if the object currently being stored has already been written out; if so, `storeOn()` simply writes out the character "@" followed by the number associated with the object in the `storeOnDict`. If the current object is not found in the `storeOnDict`, the virtual function `storer(ostream&)` is called to write out the object.

Every `storer()` function first calls the `storer()` function of its base class, with the result that the `storer()` function of the root class, `Object::storer()`, is eventually called. `Object::storer()` assigns an object number to the current object, constructs an association for the current object and its number, and adds this to the `storeOnDict`. It then uses the `isA()` function to obtain a pointer to the object's class descriptor and checks to see if any other instances of this class have been stored previously. If so, `Object::storer()` writes out the character "#" followed by the class number found in the class descriptor. If this is the first instance of this class to be stored, `Object::storer()` assigns it a class number, which is saved in the class descriptor, and then writes out a ":" followed by the name of the class and the class version number (obtained from the class descriptor).

After calling the `storer()` function of its base class, each `storer()` function writes out its member variables before returning, causing member variables to be written starting with the highest class in the hierarchy and ending with the class of the current object. Member variables that are fundamental types (e.g. `int`, `float`, `char*`, etc.) are written out directly. Member variables that are themselves class objects, or pointers to class objects, are written out by calling `storeOn()` recursively.

Thus, executing the statement `aForm.storeOn(out)`, where `aForm` is the data structure shown in Fig. 5, produces the following on the output stream `out` (reformatted to improve readability):

```

:Form.17{
  :Dictionary.4369{16 2
    :Assoc.273{
      :String.4369{12"Description"}
      :AlphaField.273{@4
        :Rectangle.17{
          :Point.17{60 20 }
          #7{100 30 }}
        #4{18"large blue widget"}}}
      #3{
        #4{12"Part Number"}
        :NumberField.273{@11
          #6{#7{10 20 }#7{50 30 }}
          123654}}}
    :OrderedCltn.4369{16 2@12 @5 }}

```

To help the reader see the relationship between the storeOn output and the aForm data structure, the class numbers assigned by storeOn() are shown in parenthesis following the class names in Fig. 5 (e.g. String (#8)), and the object numbers assigned by storeOn() for multiply-referenced objects are shown preceded by the character "@".

The inverse operation, readFrom(istream&), reads a character stream produced by storeOn(), reconstructs the object in the free store or in an area optionally provided by the caller, and returns a pointer to it. When readFrom() is called, it first encounters the class name of the object to be read, which it converts to a pointer to a function that constructs objects of that class by reading the input stream. This conversion is accomplished by using the class name to look up the class descriptor object for the class, which contains a pointer to the class's object reader() function, in a global instance of class Dictionary named classDictionary.

OOPS uses C++ static constructors to create the classDictionary. These static constructors thread the class descriptors onto a global linked list called allClasses. After all the static constructors have been executed, but still before main() is entered, the OOPS initialization routine constructs the classDictionary and adds to it an association for each class on the allClasses list. The key of each association is a String containing that class's name, and the value is the class's descriptor object.

When readFrom() is called to read the first object in a data structure, it constructs an instance of an OrderedCltn named readObjTable, which serves to translate object references read from the input stream into object pointers. Similarly, readFrom() constructs another instance of an OrderedCltn named readClassTable, which it uses to translate class numbers read from the input stream into class descriptor pointers. readFrom() looks at the next character on the input stream. If it is an "@", a reference to an existing object is to be read, so the object's number is parsed, used as an index to the readObjTable, and the object pointer found there is returned to the caller. If the next character is a ":", the following class name and class version number are read. The class name is used to look up the class descriptor object in the classDictionary, the class descriptor is added to the readClassTable, and the version number is compared to the one found in the class descriptor. A mismatch is considered an error – this detects attempts to read obsolete objects from old disk files. If the next character is a "#", the

number of a previously encountered class is read next, and the class descriptor object is found by using the class number as an index to the `readClassTable`.

Once the class descriptor has been found by either of these two methods, storage is allocated for the object (if needed), and a pointer to the object is added to the `readObjTable`. Next, the `reader(istream&,Object&)` function for the class is called via the pointer to it kept in the class descriptor.

The `reader()` function consists of a single statement that calls a constructor function with the input stream and object pointer as arguments. If the base class has a similar constructor, it is passed these same arguments; then the constructor reads the information written to the stream by the `storer()` function and completes construction of the object, calling `readFrom()` recursively to read member variables that are class objects or pointers to class objects.

One situation that this scheme does not handle properly arises when a class contains a member variable that is a class object, and a member function returns a pointer to such a member variable that somehow gets stored in another object. If `storeOn()` is used on a data structure containing both objects, the pointer to the class object member variable may be encountered and stored before the object which it is a member of is stored. When `readFrom()` processes the resulting stream, a reference to an existing object will be encountered at the point where the class object member variable should be read. The existing object cannot be moved to its proper place in the member variable of the object being constructed without relocating all pointers to it, and `readFrom()` keeps no record of where these pointers are, so it announces the error and aborts. This action is appropriate since it results from returning a pointer to a member variable, a poor programming practice that violates the principle of encapsulation.

The OOPS Object I/O facility requires that `storer()`, `reader()`, and constructor functions be implemented for each class. Other object-oriented programming languages, such as Smalltalk-80 and Objective-C<sup>6</sup>, have a more convenient object I/O mechanism that does not require the programmer to write special functions for each class. The Objective-C preprocessor, for example, produces a table that encodes the type of each member variable. The problem with this approach is that C or C++ declarations do not always contain sufficient information. For example, a variable declared as a `char*` may point to a null-terminated character string or to a single character, in which case the preprocessor must make a possibly incorrect assumption, or the programmer must supply the missing information in a comment. Such an approach was nonetheless considered during the design of OOPS, but would have required either modifying the C++ translator or writing yet another preprocessor. The approach used was chosen because the required functions are almost always short and easy to implement, and can sometimes be inherited from the base class. Furthermore, the `storer()` and `reader()` functions can often take advantage of class-specific information to reduce the size of the external representation. For example, class `Set` is implemented using a hash table that, to assure efficient searching, is never more than 50% loaded. The object I/O functions for class `Set` do not output the unoccupied entries in the table, thereby realizing significant savings.

## PROCESSES

The main design goal of incorporating processes into OOPS was to support a moderate number of coroutines that could be scheduled for execution in response to external events such as keyboard input, mouse input, and asynchronous I/O completion. OOPS

processes are functionally similar to those of Smalltalk-80, but the implementation borrows some key ideas from the task library that is distributed with the AT&T C++ Translator<sup>7</sup> and from BLISS-11<sup>8</sup>.

As an example, consider a process that responds to keystrokes typed at the standard input terminal. Class `Process` serves as the base class for a class with a constructor function that gets keystroke event objects from a queue and processes them in an endless loop. The keystroke event objects are placed in the queue by a UNIX signal handler function (not shown).

```
class KeybdProcess : public Process {
public:
    KeybdProcess(const char* name,      // process name
                 EventQueue&   keystrokeQ);
};

KeybdProcess::KeybdProcess(const char* name,
                           EventQueue& keystrokeQ)
    : (name, 512 /*stack size*/, 1 /*priority*/)
{
    PROCESS_FORK;
    KeybdEvent* keystroke;
    while (YES) {
        keystroke = (KeybdEvent*)keystrokeQ.nextEvent();
        // process keystroke here ...
        delete keystroke;
    }
}
```

Thus, the `new` operator creates and starts an instance of a process:

```
EventQueue& rawin = *new EventQueue;
Process* p = new KeybdProcess("keyboard input", rawin);
```

The member variables of class `Process` hold the context of a process: the address, length, and current top of the process's execution stack, and its name, priority (0 to 7), and state (`SUSPENDED`, `RUNNING`, or `TERMINATED`). The member variables of classes derived from class `Process` also become part of the process context. When a new process is created, storage for the process context and stack is allocated and initialized and the user's code in the derived class's constructor begins execution. The `PROCESS_FORK` statement links the new process on the OOPS process scheduler's run list, and it will be executed when it is the highest priority process. Note that processes are never pre-empted; a process must explicitly relinquish control, usually by blocking on a Semaphore object, before another process is run, even if it has a higher priority. This simplifies data sharing among OOPS processes, and is in fact necessary because C run-time library routines such as `malloc()`, the memory allocator, are not reentrant.

Process synchronization is handled as it is in Smalltalk-80, by `signal()` and `wait()` operations on instances of class `Semaphore`. Processes can communicate by sending objects to one another via instances of class `SharedQueue`. A `SharedQueue` is a

fixed length queue of object pointers (an instance of class `Arrayobid`) with access controlled by a pair of Semaphore objects. One of the semaphores blocks writing processes when the queue is full, the other blocks reading processes when the queue is empty.

In the example, class `EventQueue` is a derived class of class `SharedQueue` that allows only Event objects to be placed on the queue. Event objects are distinctive in that they are allocated in a dedicated area of memory by an reentrant storage manager, thus permitting them to be constructed by UNIX signal handlers, which are executed asynchronously in response to external events. The `EventQueue` member function `nextEvent()` copies events into memory allocated using `new`, frees the storage used for the queued object so it may be reused by signal handlers, and returns the copy of the event to the calling process.

Under 4.2BSD UNIX, OOPS uses the `sigblock()` system call to disable signals while shared data structures such as the scheduler's lists of running processes are being modified. MASSCOMP's Real-Time UNIX<sup>TM</sup> (RTU), the operating system under which OOPS was originally developed, provides Asynchronous System Traps (ASTs) in addition to signals. ASTs are like 4.2BSD signals, but can also pass a parameter to the handler routine and can be delivered at different priorities. Under RTU, OOPS uses ASTs instead of signals, and uses the `setpri()` system call to disable ASTs when necessary. Certain events, such as asynchronous I/O completion, that can only generate signals are converted to ASTs by the signal handler. OOPS can do nothing with signals under UNIX System V.2 because they are not reliably delivered, and asynchronous I/O is not available. OOPS processes can still be used, but they cannot respond to external events.

The OOPS implementation of processes requires two functions that must be written in assembly language because they manipulate the execution stack: `Process::create(void**)` and `Process::exchj()`. The `create()` function initializes a new stack area by copying enough of the current stack to it to include the parameters passed to the new process's constructor, and leaves it in a state such that a subsequent call to `exchj()` will start the new process. The VAX<sup>TM</sup> implementation of `create()` is 16 instructions long. The `exchj()` function saves all registers on the current stack, switches the stack pointer to the stack of the process to be resumed, restores the registers saved there by a previous call to `exchj()` or `create()`, and returns. The VAX implementation of `exchj()` requires 5 instructions.

## DIFFERENCES FROM SMALLTALK-80

OOPS efficiently implements much of the basic functionality of Smalltalk-80 within a UNIX/C environment. The tradeoff is that it cannot provide some of Smalltalk's more sophisticated features. One major difference is that Smalltalk-80 treats everything uniformly as objects, including fundamental data types (such as integers and characters) and blocks of code. Thus, it is possible to add an integer to a collection, or pass a block of code as a parameter to a function. OOPS handles the former by providing classes such as `Integer` and `Float`, but does not implement arithmetic for these classes because it would encourage their relatively inefficient use.

OOPS has no way to deal with blocks of code as objects, however, so it does not implement the many Smalltalk-80 functions that use blocks of code as parameters. This primarily impacts how iteration over the objects in a collection is programmed. In

Smalltalk-80, the code to process an object in a collection is passed as a block parameter to a member function of the collection class that executes the block for each object in the collection, thus hiding the mechanism for iterating over the objects in the collection in the function. For example, the `Form::fillIn()` function described previously could be written in Smalltalk-80 as:

```
aForm do: [ :aField | aField fillIn ]
```

In C++, the iteration must be programmed explicitly as a loop, and the collection iteration mechanism can be hidden by providing a class (e.g. `DoCollection`) to maintain the state of the iteration and with a member function `Object* next()` to return the "next" object to be processed:

```
Form aForm = ...;           // an instance of class Form
DoCollection do(aForm);     // maintains state of iteration over aForm
Field* aField;
while (aField=(Field*)do.next()) aField->fillIn();
```

This solution is obviously not as elegant as using a block as a parameter in Smalltalk-80.

Another major difference between Smalltalk-80 and OOPS is that Smalltalk-80 provides automatic garbage collection, so users do not have to worry about deleting the objects they create – they are freed automatically when they are no longer referenced. However, an OOPS user must be aware of object storage management, and this is a source of serious programming errors. For example, a user may declare an object that is a local variable of a block, then add this object to a collection object that is used outside the block. When the block exits, storage for the local object is deallocated, but a pointer to the nonexistent object is still stored in the collection where it may be used to overwrite another object later allocated to the same space. To help track down this type of problem, a special debugging version of the memory allocator is used that logs all allocations and deallocations and checks its data structures for consistency. Adding automatic garbage collection to C++ would eliminate this type of problem, but the added overhead would be unacceptable for many applications.

Other differences between Smalltalk-80 and OOPS arise from Smalltalk-80's implementation as an interpreted language, whereas C++/OOPS is compiled. Thus, Smalltalk-80 programs have more information about themselves available at execution time and can be modified dynamically. For example, in Smalltalk-80 any object can be interrogated to see if it implements a specified function, or a new class can be created at execution time. These capabilities are gained at the expense of efficiency, however.

In Smalltalk-80, a variable may hold any class of object. In C++, the class of each variable is statically declared, and use of the variable is type-checked at compile time, which results in early detection of programming errors and makes programs easier to understand since the programmer's intended use of each variable is explicit. It can also improve efficiency since, for example, the size of a variable may be known at compile time and dynamic memory allocation avoided. Of course, much of the power of object-oriented programming results from dynamic binding, where the specific class of a variable is not known until execution time. OOPS uses C++ derived classes and virtual functions to provide this facility, and programmers can control the specificity of the variables in their programs. For example, a variable pointing to an instance of class `Set` might be declared to be a `Set*` (very specific), a `Collection*` (moderately specific), or an `Object*` (very general), whichever is most appropriate in a particular situation. In

Smalltalk-80, the class of object that a variable can hold cannot be restricted except by explicit run-time checks.

A programmer must occasionally override C++'s type checking, however. This occurs when using classes such as the collection classes that structure arbitrary objects. These classes manipulate pointers to instances of class `Object` (`Object*`), and when a client program accesses any of these instances, it must be by means of an `Object*` pointer obtained by calling one of the class's member functions. A Smalltalk-80 program can attempt to invoke any function on any object – a run-time error will occur if the object does not implement the function. A C++ program, however, cannot invoke the member functions defined in a more specific class via a pointer of type `Object*` – a cast must be used to coerce the type of the pointer to that of the more specific class, and disaster will result if the object is not actually of the expected class. Often, as was the case in the form example presented earlier, the client program can utilize C++ static typing to guarantee that instances of the correct class only are added to the data structure so that the cast is safe when the instances are accessed and no run-time checking is needed. For situations when static typing is not sufficient, class `Object` implements member functions for checking an object's class at run time.

## CONCLUSION

The OOPS class library has been used to implement a prototype forms management package for ASCII display terminals. The package consists of about 7000 lines of C++ code and required 18 programmer-weeks to design and implement, including the time required to learn object-oriented programming, the C++ programming language, and the OOPS class library. A significant portion of this time was also spent in dealing with problems in the C preprocessor, early versions of the C++ translator, and the C compiler. The following observations are made based on this experience.

C++ is a practical language for object-oriented programming, and large, general-purpose class libraries such as OOPS can be constructed with it. The OOPS class library offers many of the basic features of Smalltalk-80, is relatively efficient, and can be used with existing C libraries for input/output, graphics, communications, database management, and numeric computation.

Overall, programming with C++ and the OOPS class library seems less error-prone than with C due to the stronger type checking performed by C++, encapsulation, and the increased number of opportunities to reuse working code from existing classes. The most common serious errors experienced are those of improper storage management, as previously described.

Object-oriented programming and C++ were relatively difficult to learn, due in large part to a lack of suitable training materials and also because we had to use the most advanced features of C++ immediately since a class library was not available. The extra effort was worthwhile; however, due to the improvement in programming productivity we experienced.

## ACKNOWLEDGEMENTS

The forms management system mentioned in the previous section was designed and implemented by Sandy Orlow, the intrepid first OOPS user, who provided much valuable feedback. The organization and readability of this paper were greatly improved thanks to the constructive comments of Perry Plexico and Bjarne Stroustrup.

## APPENDIX A: AN EXAMPLE OOPS PROGRAM

The following C++ program is intended to give an idea of how to use the OOPS class library and what it can do. The program builds a data structure that contains dates and the names of people born on those dates. When a date is entered, it is looked up in the database and the names of the people born on that date, if any, are printed alphabetically in response.

The program is divided into four files: **DateDict.h**, which contains common declarations needed to compile the other three files, **DateDict.c**, which implements functions for building and manipulating the database, **createdict.c**, which creates an example database and saves it in a disk file, and **datedict.c**, which reads in this file and processes user queries.

The file **DateDict.h** first includes the declarations for the OOPS library classes used by the program:

```
#include "Dictionary.h"           // associative arrays
#include "SortedCltn.h"           // sorted collections
#include "String.h"               // character strings
#include "Date.h"                 // calendar dates
```

Next, class **DateDict** is defined as a specialized version of the OOPS library class **Dictionary**. Functions for initializing the data structure, adding date/name pairs, and looking up the names associated with a particular date are specified:

```
extern Class class_DateDict;      // descriptor for class DateDict
class DateDict : public Dictionary { // specialized version of class Dictionary
public:
    DateDict() {}                 // constructor
    DateDict(istream&, DateDict&); // constructor used by readFrom()
    virtual const Class* isA();    // returns pointer to class_DateDict
    void addName(const Date& birthdate, const String& name);
    SortedCltn& lookup(const Date& birthdate);
};
```

The file **DateDict.c** contains the implementation of class **DateDict**. It includes the declarations from the file **DateDict.h**, calls the OOPS **DEFINE\_CLASS** macro to create some data structures and functions required by the OOPS class library, defines a special constructor function needed for reading the database from a file, and implements the functions **addName()** and **Lookup()**:

```

#include "DateDict.h"
DEFINE_CLASS(DateDict,Dictionary,1, NULL,NULL);

DateDict::DateDict(istream& strm, DateDict& where) : (strm,where)
{
    this = &where;
} // constructor used by readFrom()

void DateDict::addName(const Date& birthdate, const String& name)
// Add a birthdate and name to the dictionary
{
    SortedCltn* list; // the list of names associated with this date
    if (includesKey(birthdate)) list = &lookup(birthdate);
    else { // date is not yet in dictionary
        list = new SortedCltn; // create a new sorted collection
        addAssoc(birthdate,*list); // add date and empty list to dictionary
    }
    list->add(name); // add name to list for this date
}

SortedCltn& DateDict::lookup(const Date& birthdate) {
// Look up the sorted collection of names associated with a date.
    return *(SortedCltn*)atKey(birthdate);
}

```

The file **createdict.d** is a program that creates an empty **DateDict**, adds some date/name pairs to it, and saves it on a file named **datedict.dat**:

```

#include <osfcn.h> // declarations of standard C library functions
#include "DateDict.h"
main()
{
    DateDict d; // create an instance of a DateDict named d
// Add some birthdates to the DateDict d
    d.addName(*new Date(11,"Nov",48), *new String("Smith, Mary"));
    d.addName(*new Date(11,"Nov",48), *new String("Doe, John"));
    d.addName(*new Date(10,"May",47), *new String("Jones, Bill"));
    ostream out(creat("datedict.dat",0664)); // protection mode 0664
    d.storeOn(out); // store DateDict d on file "datedict.dat"
}

```

Finally, the file **datedict.c** is a program that opens the file **datedict.dat**, reads the **DateDict** stored there into memory referenced by the identifier **d**, and processes queries entered by the user:

```

#include <osfcn.h>           // declarations of standard C library functions
#include <fcntl.h>           // declarations of open() and fcntl() flag values
#include "DateDict.h"
main()
{
    istream in(open("datedict.dat", O_RDONLY));
    DateDict& d = *(DateDict*)readFrom(in, "DateDict");
    Date date;               // an instance of Date to hold the user's query
    while (YES) {
        cout << "\nEnter date: ";    // prompt the user for a date
        cin >> date;                  // read the date entered by the user
        if (cin.eof()) exit(0);       // exit if user indicates end of input
        // if the date is found, print out the list of names associated with it
        if (d.includesKey(date)) cout << d.lookup(date);
        else cout << "not found";
    }
}

```

Here is an example run of this program:

```

Enter date: 11/11/48
Doe, John
Smith, Mary
Enter date: May 10, 1947
Jones, Bill
Enter date: 10-MAR-85
not found

```

## REFERENCES

1. A. Goldberg and D. Robson, *Smalltalk-80 The Language and its Implementation*, Addison-Wesley, Reading, Massachusetts, 1983.
2. T. Kaehler and D. Patterson, 'A Small Taste of Smalltalk', *Byte*, 11, No. 8, 145-159 (August 1986).
3. D. Robson, 'Object-Oriented Software Systems', *Byte*, 6, No. 8, 36-48 (August 1981).
4. J. C. Althoff Jr., 'Building Data Structures in the Smalltalk-80 System', *Byte*, 6, No. 8, 230-278 (August 1981).
5. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Massachusetts, 1986.
6. B. Cox, *Object-oriented Programming*, Addison-Wesley, Reading, Massachusetts, 1986.
7. B. Stroustrup, 'A Set of C++ Classes for Co-routine Style Programming', *UNIX System V AT&T C++ Translator Release Notes*, AT&T, 1985.
8. W. Wulf, *BLISS-11 Programmer's Manual*, Digital Equipment Corporation, Maynard, Massachusetts, 1972.

Table I. OOPS Classes

<b>Object</b>	– Root of the OOPS Class Inheritance Tree
<b>Bitset</b>	– Set of Small Integers (like Pascal's type SET)
<b>Class</b>	– Class Descriptor
<b>Collection</b>	– Abstract Class for Collections
<b>Arraychar</b>	– Byte Array
<b>String</b>	– Character String
<b>Arrayobid</b>	– Array of Object Pointers
<b>Bag</b>	– Unordered Collection of Objects
<b>Set</b>	– Unordered Collection of Non-Duplicate Objects
<b>Dictionary</b>	– Set of Associations
<b>IdentDict</b>	– Dictionary Keyed by Object ID
<b>SeqCltn</b>	– Abstract Class for Ordered, Indexed Collections
<b>Heap</b>	– Min-Max Heap of Object Pointers
<b>LinkedList</b>	– Singly-Linked List
<b>OrderedCltn</b>	– Ordered Collection of Object Pointers
<b>SortedCltn</b>	– Sorted Collection
<b>Stack</b>	– Stack of Object Pointers
<b>Date</b>	– Gregorian Calendar Date
<b>Float</b>	– Floating Point Number
<b>Fraction</b>	– Rational Arithmetic
<b>Link</b>	– Abstract Class for LinkedList Links
<b>Linkobid</b>	– Link Containing Object Pointer
<b>Process</b>	– Co-routine Process Object
<b>LookupKey</b>	– Abstract Class for Dictionary Associations
<b>Assoc</b>	– Association of Object Pointers
<b>AssocInt</b>	– Association of Object Pointer with Integer
<b>Integer</b>	– Integer Number Object
<b>Nil</b>	– The Nil Object
<b>Point</b>	– X-Y Coordinate Pair
<b>Random</b>	– Random Number Generator
<b>Rectangle</b>	– Rectangle Object
<b>Scheduler</b>	– Co-routine Process Scheduler
<b>Semaphore</b>	– Process Synchronization
<b>SharedQueue</b>	– Shared Queue of Objects
<b>Time</b>	– Time of Day
<b>Vector</b>	– Abstract Class for Vectors <sup>1</sup>
<b>BitVec</b>	– Bit Vector
<b>ByteVec</b>	– Byte Vector
<b>ShortVec</b>	– Short Integer Vector
<b>IntVec</b>	– Integer Vector
<b>LongVec</b>	– Long Integer Vector
<b>FloatVec</b>	– Floating Point Vector
<b>DoubleVec</b>	– Double-Precision Floating Point Vector
<b>ComplexVec</b>	– Complex Vector

<sup>1</sup>The Vector classes are experimental at the time this paper is being written.

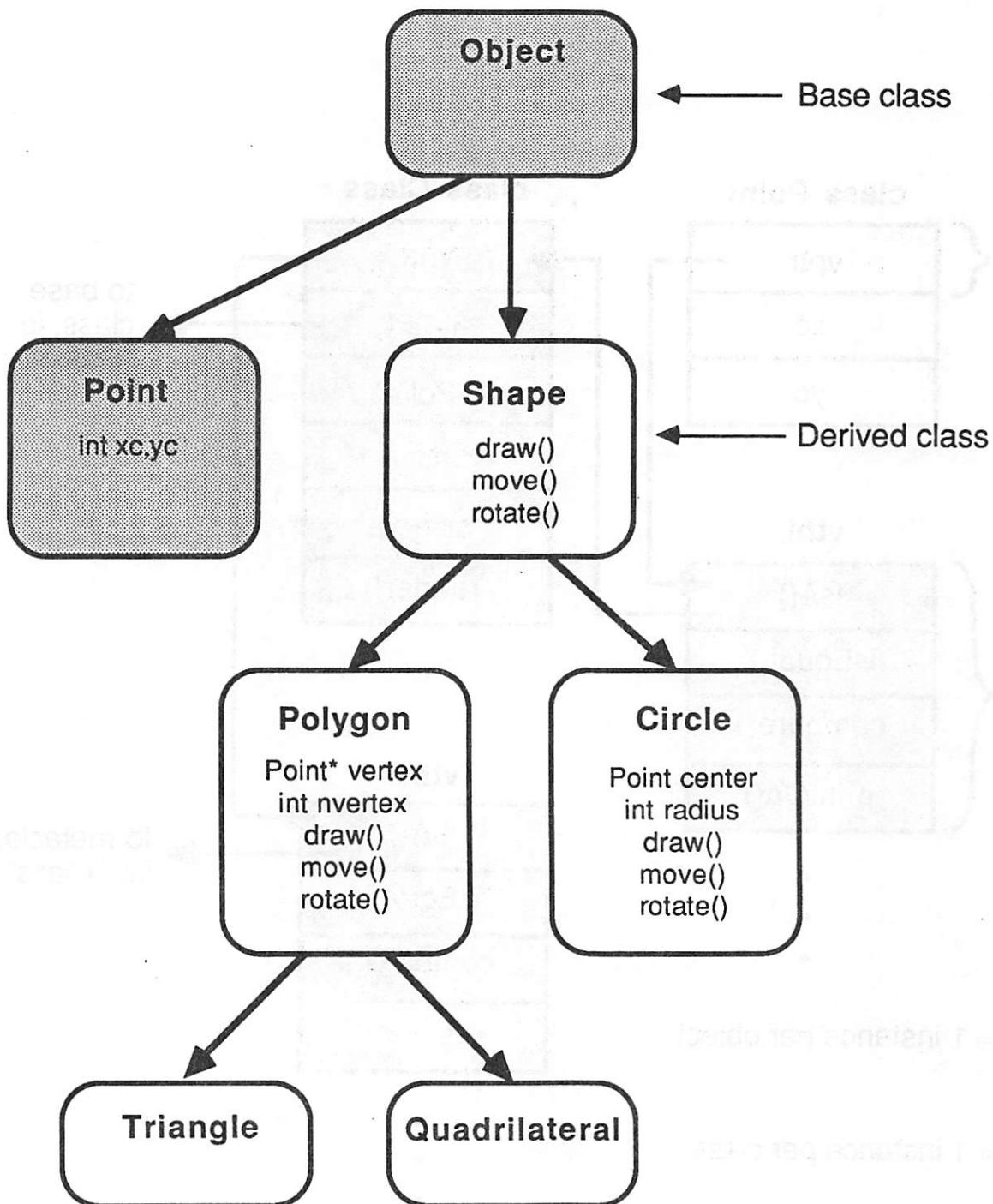


Figure 1. Class hierarchy for shape example, showing the relation between classes and some of their member variables and functions. Shaded rectangles represent classes that are part of the OOPS library.

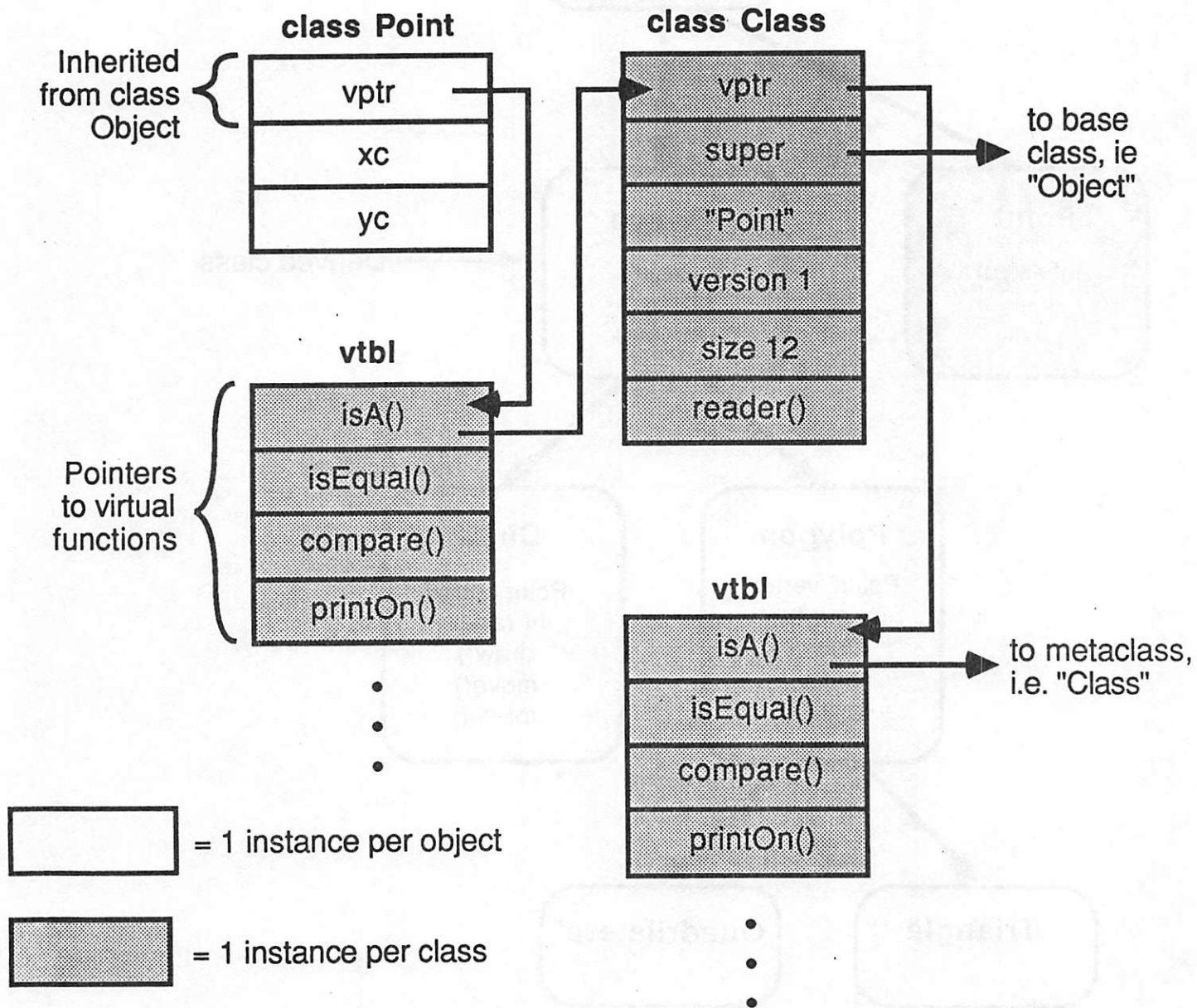


Figure 2. Data structure for an instance of class `Point`

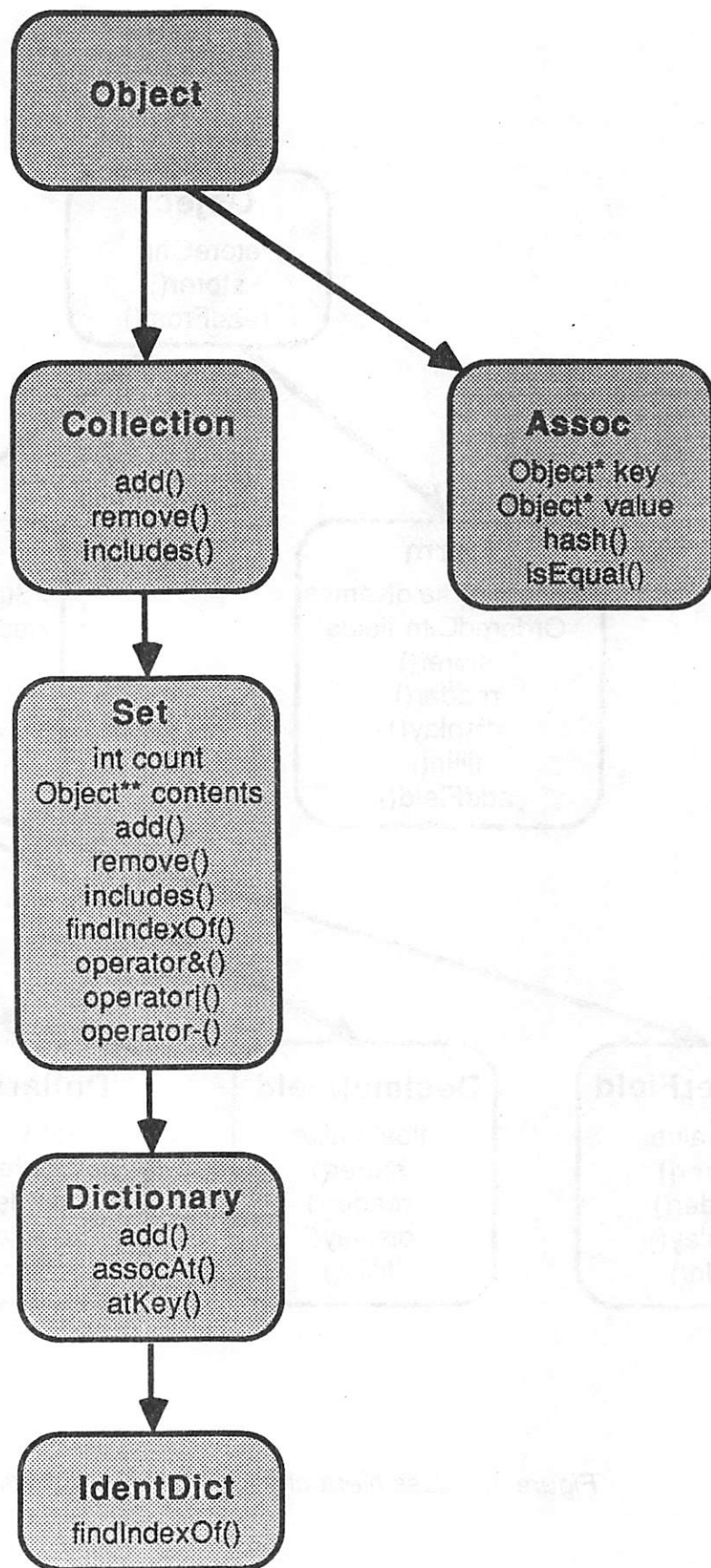


Figure 3. A portion of the OOPS class hierarchy showing the classes used in the implementation of classes Dictionary and IdentDict.

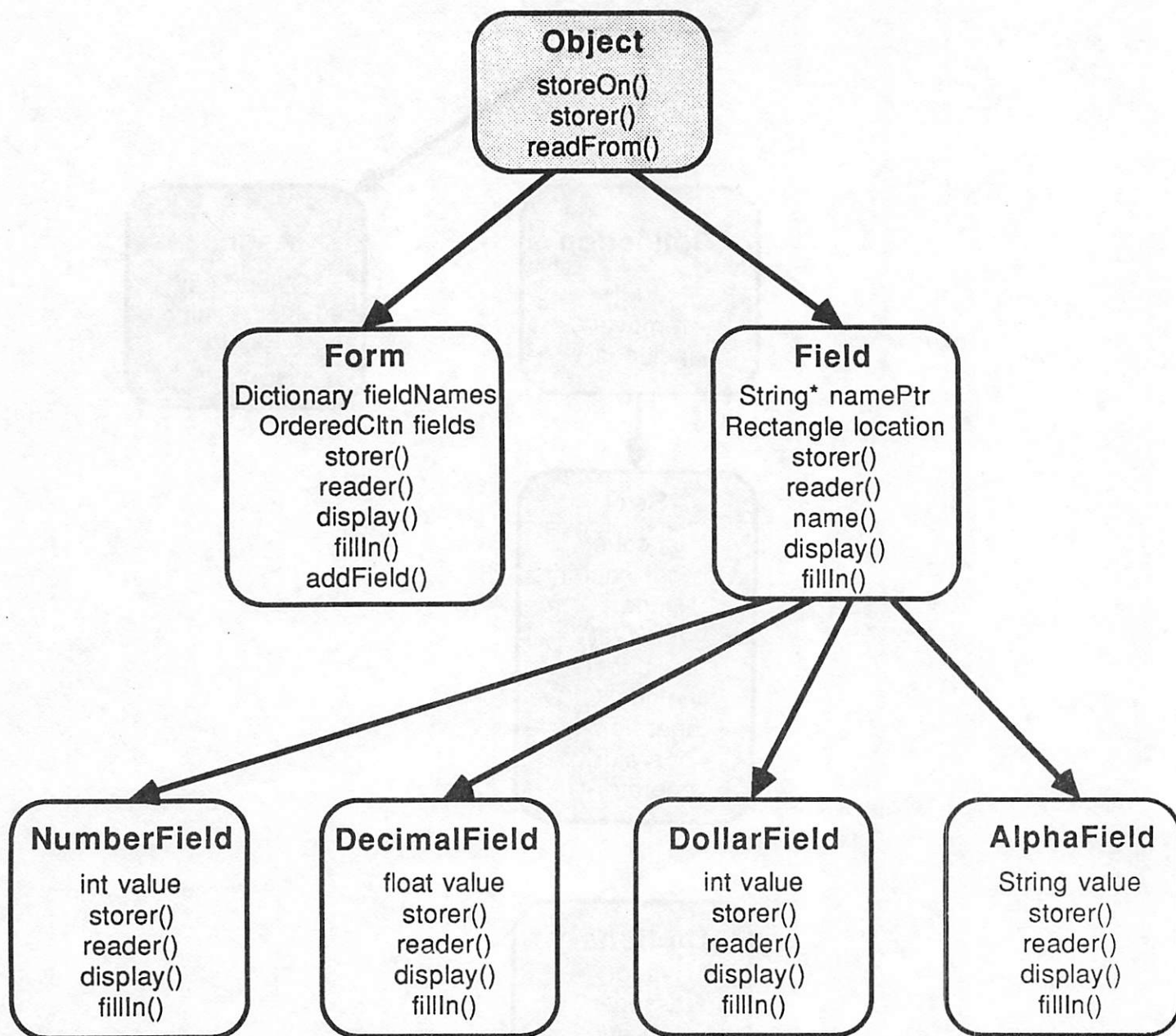


Figure 4. Class hierarchy for example form management system

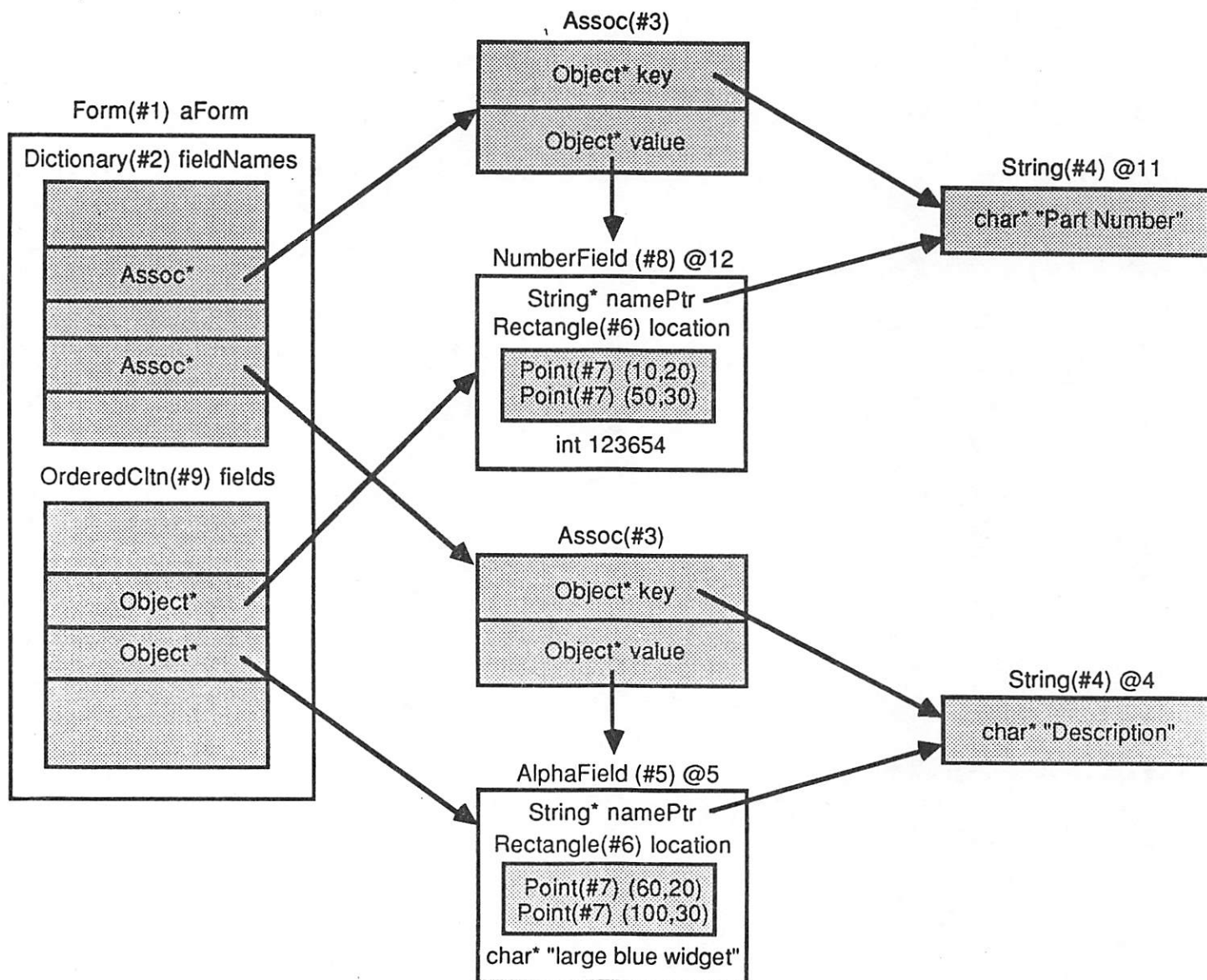
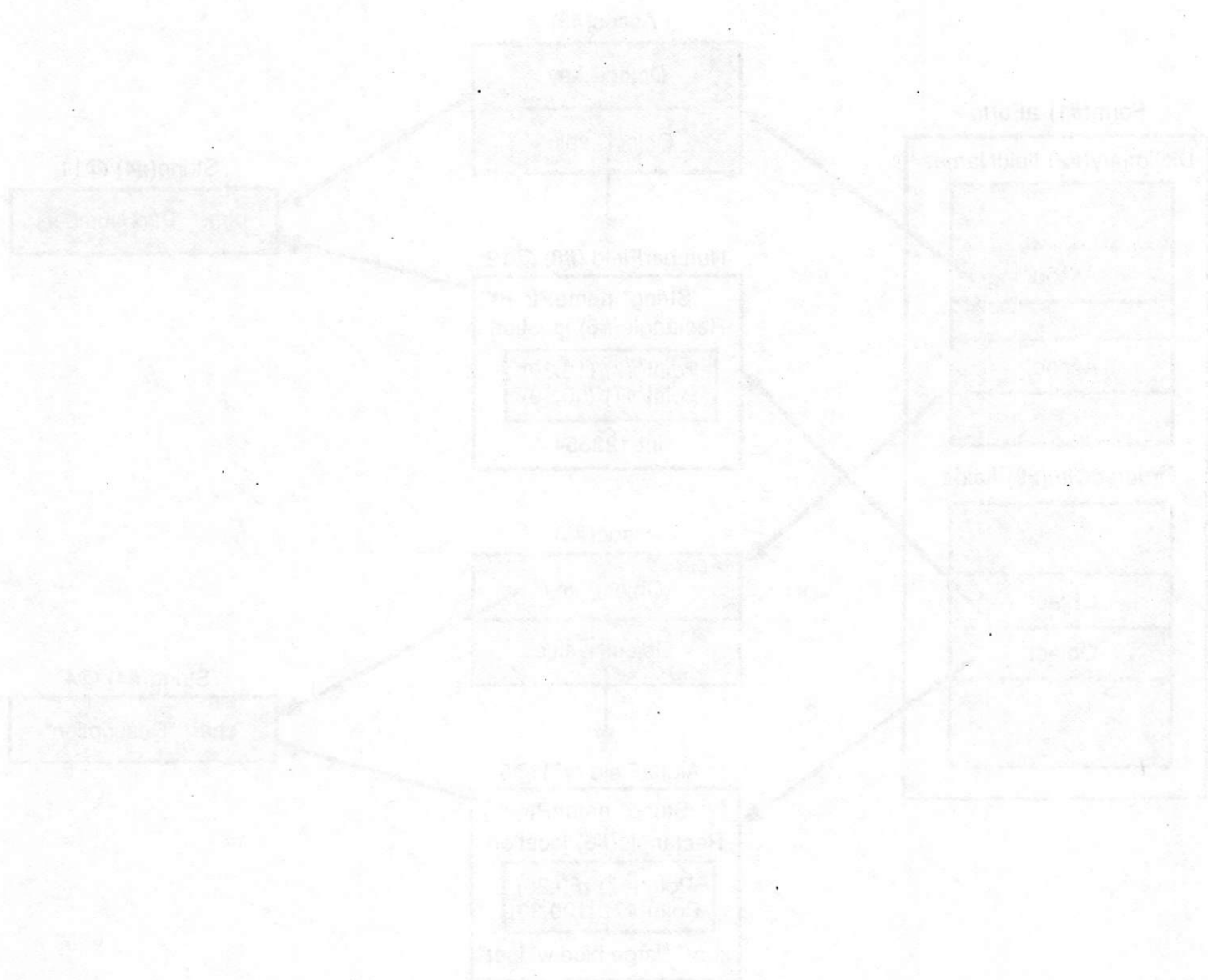


Figure 5. An instance of a Form with two fields, a NumberField and an AlphaField. OOPS library classes are shown as shaded rectangles. Numbers preceded by a "#" are class numbers, and numbers preceded by a "@" are object numbers.



OBJECT-ORIENTED CLASS LIBRARY FOR C++

Ken Fuhrman  
Ampex Corporation  
Wheat Ridge, Colorado 80033

ABSTRACT

This class library provides a set of classes to support object-oriented programming. The class library allows the design of persistent dynamic objects. It is meant to provide a method to construct dynamic objects for data structure design. A set of classes and operations are provided for the construction, editing, and traversal of objects. The goal is to provide a standard object-oriented approach to data structure design.

OUTLINE

- 1) Goals of the Class Library
- 2) Overview of the Object-Oriented Class Library
- 3) The Basic Class Hierarchy
- 4) Object Properties
- 5) Object References
- 6) Object Construction
- 7) Object Operations
- 8) An Example of the Object-Oriented Class Library
- 9) Applications
- 10) Conclusion

## 1) Goals of the Class Library

An Object-Oriented Class Library for C++ was developed for a project at Ampex. It was designed to support the implementation of the User Interface and the application software that deals with the manipulation of graphical objects. This class library provides an extensible method for designing objects based around the basic class hierarchy provided in the library.

The following goals were identified when designing the Object-Oriented Class Library:

- o Provide support for persistent objects.
- o Provide support for constructing dynamic objects.
- o Allow the design of arbitrarily complex objects.
- o Reduce design complexity of complex dynamic objects.
- o Optimize for size and speed in traversal of dynamic objects.
- o Reduce code redundancy.
- o Develop a consistent methodology for object design.
- o Provide a basic set of operations for the construction of dynamic objects.
- o Support an extensible method for further additions to the library.
- o Provide an environment for designing application (user) objects.

The Object-Oriented Class Library does not provide any specific application objects such as OOPS does. Thus, objects such as strings or integers are not inherently provided, although it is intended that a user of the class library can easily build such objects. What is provided is a set of classes and operations that allow the design of persistent and dynamic objects. There are a full set of operations for the construction, traversal and editing of these objects. It is the responsibility of the user to design specific application objects from the class hierarchy.

In designing dynamic data structures it was noticed that individual programmers implemented varying sets of operations in order to construct, traverse and edit dynamic data structures. Much of the code that was associated with the object was concerned with construction, traversal and editing. Also, each object typically had different methods for saving and restoring from files. Since many of the applications this project is concerned with deal with User Interfaces and graphics applications which require dynamic data structures, a unified approach for an object design methodology was desired.

## 2) Overview of the Object-Oriented Class Library

The Object-Oriented Class Library is designed around five basic classes that are organized into a class hierarchy and a set of operations that allow the construction, traversal and editing of objects. Since the Object-Oriented Class Library does not provide any specific objects, it is required for the user to create new objects from the basic classes.

There are two methods for creating objects. The first is to derive simple objects directly from one of the basic classes. These user created classes are called user classes. The second way is to construct dynamic objects from other simple or dynamic objects. The concept of construction is integral to the Object-Oriented Class Library and many of the basic set of operations are provided for the construction of more complex objects. The Object-Oriented Class Library is suited for creating dynamic objects that are list or tree based, although more complex data structures can be constructed.

All objects are persistent, even dynamically constructed objects. All objects derived from the basic classes are self-decoding in terms of size. The size of an object is determined at run-time thus variable sized objects can be persistent. Some other capabilities of the Class Library are exception handling, object properties and object references.

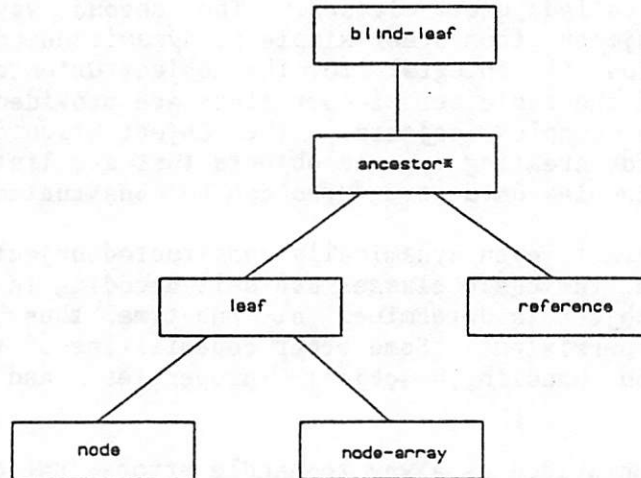
Exception handling is provided as a way to handle errors that occur in the object operations. Users of the class library can also call the exception handler to deal with errors in the user class operations.

The concept of properties is provided for objects. Properties, which are objects themselves, can be attached to objects via a property list. These properties can then be interpreted by the object.

Object references are a method to point to other objects within a complex object.

### 3) The Basic Class Hierarchy

The Object-Oriented Library is designed around 5 basic classes that the user can derive from to create simple objects. These classes are organized into a class hierarchy. The basic classes that the user can derive from are: `Blind_Leaf`, `Leaf`, `Reference`, `Node`, and `Node_Array`. The class hierarchy is shown below:

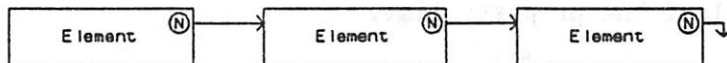


### The Class Element and Lists

There is a class called `Element` that is the base class for class `Blind_Leaf` and it is not shown in the basic class hierarchy. The class `Element` provides a next pointer and previous pointer that allows the construction of doubly-linked lists. Lists are the basic data structure that construction operations are provided for. The class `Element` provides a set of member operations and functions that operate on lists. Many of the operations provided for the construction of objects use the class `Element` member functions.

Normally a user of the Class Library will not derive a user class from the class `Element`, although this is not prevented. If a user class is derived from class `Element` then an instantiation of that user class is not a persistent object. The reason the class `Element` is provided for deriving user classes is to allow deriving simple classes that only need list operations and are not required to be persistent. Often

temporary data structures that are list based are required, thus they may be derived from class Element.



### The Class Blind Leaf

The class `Blind_Leaf` provides the data field required for an object to be self-decoding in size. The size field represents the number of bytes of contiguous space that a simple object occupies. The size field is set by the constructor provided in class `Blind_Leaf`, also the size field can be passed in as a parameter to the constructor. All constructors for the 4 classes derived from `Blind_Leaf` allow the size field to be passed in. This allows variable sized objects to be created at run-time.

There is a type field provided for the application object type. This type field was originally used by some user classes to decode at run-time the type of object that was dealt with.

Normally dynamic binding is used by way of virtual functions to decode the type of an object. There is a virtual function `type()` provided in class `Blind_Leaf` that can be used to determine the object type.

The class `Blind_Leaf` is provided for objects that only need persistency and do not require the other capabilities provided in the other 4 classes. The class `Blind_Leaf` derives its name from the fact that it lacks the ancestor pointer, provided by class `Ancestor`. Thus if an object of type `Blind_Leaf` is in a hierarchical data structure it is not possible to traverse back up the data structure just using the object `Blind_Leaf`, it is blind to the data structure above it. With class `Blind_Leaf` it is only possible to move to the next or previous object in a list.

### The Class Leaf

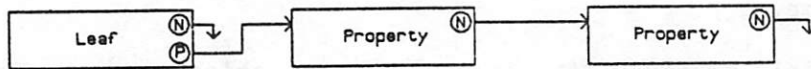
The class `Leaf` provides the property list pointer and a reference count. Properties can be attached to an object via the property list. Any object can be placed in an object's property list. In order to have a property list an object must be derived from one of the basic

classes that has class Leaf in it's class hierarchy. The reference count is used by the Reference class.

There are a set of operations provided that allow the construction, editing and traversal of the property list.

Properties are not interpreted by any of the property operations. This is left up to the user that created the object and the properties that are attached to the object. The user will typically add operations to an object that use objects in the property list and interpret there meaning.

The class Leaf has the ancestor pointer provided by class Ancestor, thus from an object of type Leaf, in a hierarchical data structure, it is possible to move to other parts of the data structure using the traversal operations.



### The Class Reference

The class Reference provides a reference pointer that can point to any other object. The class Reference is an "object pointer".

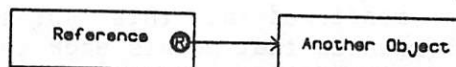
The traversal operations provided in the class Library will dereference references when traversing the object structure.

References can present a problem for the operations that delete all or part of an object. This problem is handled by the concept of a reference count that was introduced in Class Leaf. The reference count is incremented whenever an object is referenced and decremented whenever a reference is deleted that pointed to that object. Thus, only objects that have the class Leaf in their class hierarchy can be referenced.

The set of reference operations will not allow a reference to be dangling, point to an incorrect object class type or allow the object that is referenced be deleted.

Normally the class Reference is instansiated directly as an object without deriving a user class from it. But, it is possible to derive a

user class from class Reference and include additional data and operations that deal with the reference in a more specific manner.

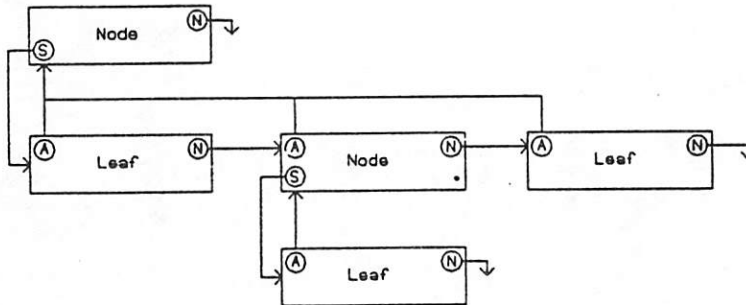


## The Class Node

The class Node introduces a data field called a substructure pointer. The substructure pointer points to a list that can contain any object. This allows hierarchical objects to be constructed.

The substructure refers to the data structure that is below a Node. There are a set of operations that deal with the substructure of an object and constructing hierarchical dynamic objects.

The class ancestor should be mentioned at this point. The class ancestor provides an ancestor pointer that points back to the node that has the object in the node's substructure. The construction and editing operations will make sure that the ancestor pointer is set up properly. The traversal operations use the ancestor pointer to move around in the data structure.



### The Class Node Array

The class `Node_Array` is similar to class `Node`. The only difference is that there is an array of substructure pointers in class `Node_Array`.

The set of operations that deal with the substructure in class `Node` are overloaded for class `Node_Array`. The class `Node_Array` operations require one additional parameter which is the index of the substructure.

### Deriving User Classes from the Basic Classes

The basic classes, except for class `Reference`, are never instantiated directly as objects. A user class is derived from one of the basic classes, additional data and member operations are normally provided for in the newly derived class. This user class can then be instantiated as a simple object.

```
class Transform_2D : public Leaf
{
    double matrix[3][3];
public:
    Transform_2D();
    rotate(double theta);
    scale(double s);
    scale(double sx, double sy);
    translate(double tx, double ty);
};
```

When designing an object, the user determines what capabilities are desired, if it is a simple object, such as a character array, then it is normally derived from class `Blind_Leaf`. If the user is intending on constructing a more complex object, such as a tree of graphical primitives then the object would be derived from the class `Node`, `Node_Array` or `Leaf`.

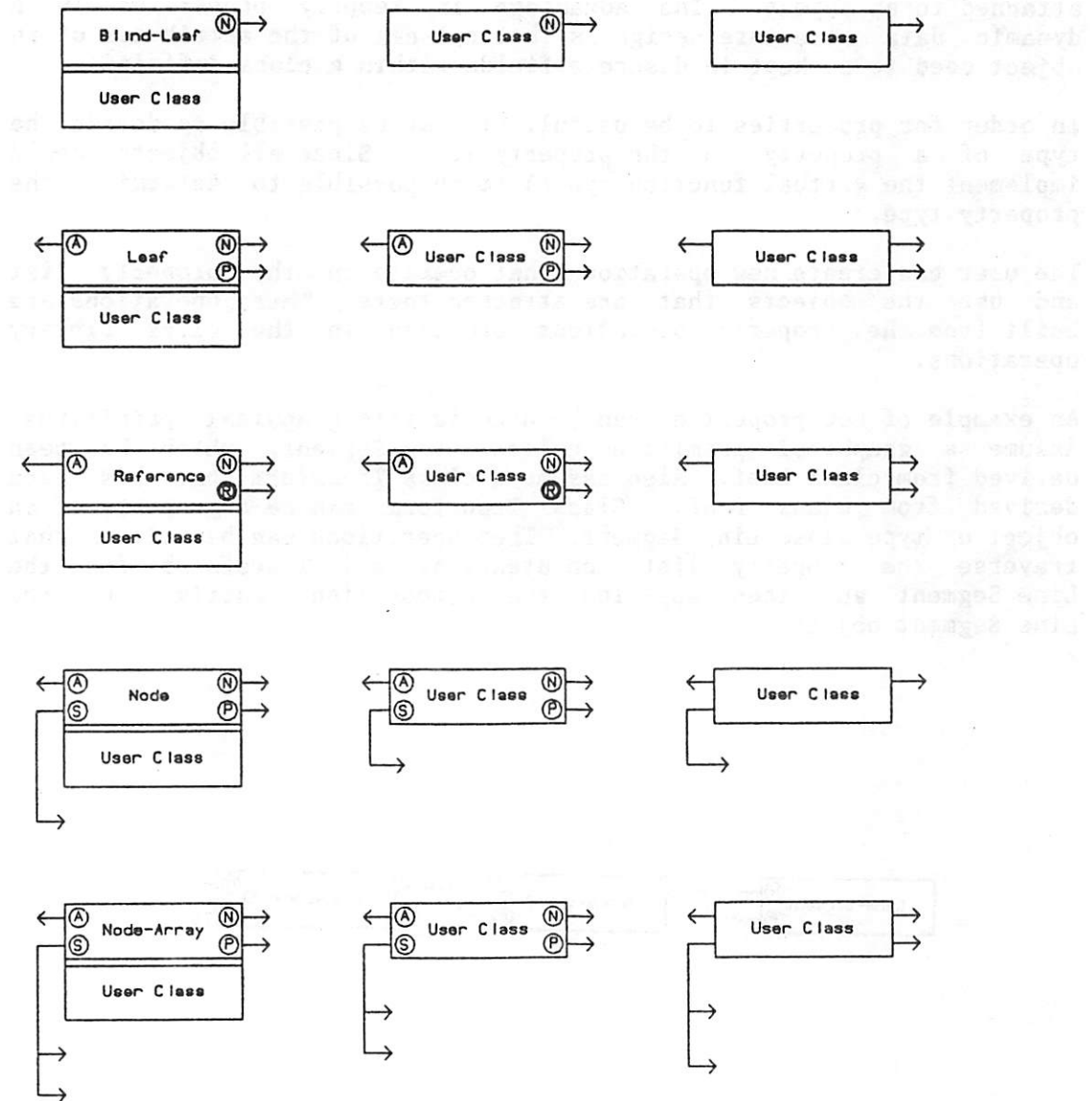
One point should be mentioned here. If a user adds pointers to other objects or data structures in a user class then in order for persistency to work the virtual function `wr_simple_obj()` must be implemented. If the user class is just a contiguous simple object then it is not necessary to implement `wr_simple_obj()` for persistency to work.

The user is adding to the Class Library hierarchy as user classes are derived. Thus the class hierarchy tree can be used to keep the user classes in perspective.

## A Graphical Notation for Objects

The following is an example graphical notation of how simple and complex objects can be drawn. A graphical notation for each basic class will be shown (simple objects). Most of the drawings in the paper were based on this graphical notation.

The basic class name is shown in the top box with the use class in the lower box, separated by a double line. The next point is a line that comes from the upper right and has an "N" next to it. The Property List pointer comes from the lower right and has a "P" next to it. The Ancestor pointer comes from the upper left and has an "A" next to it. The substructure pointer comes from the lower left and has an "S" next to it. The reference pointer is the lower left and has an R next to it.



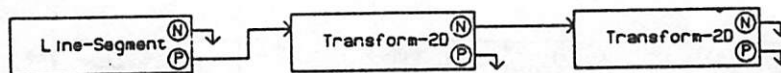
#### 4) Object Properties

Properties are an important concept in complex object design. Properties can be used to deal with various attributes that can be attached to an object. The advantage in keeping properties in a dynamic data structure design is that not all of the attributes of an object need to be kept in discrete fields within a class definition.

In order for properties to be useful, it must be possible to decode the type of a property on the property list. Since all objects should implement the virtual function `type()` it is possible to determine the property type.

The user can create new operations that operate on the property list and use the objects that are attached there. These operations are built from the property operations provided in the Class Library operations.

An example of how properties can be used is with graphical primitives. Assume a graphical primitive called `Line_Segment`, which has been derived from class `Leaf`. Also assume a class `Transform` that has been derived from class `Leaf`. Class `Transform` can be a property of an object of type class `Line_Segment`. Then operations can be written that traverse the Property list concatenating all `Transform`s for the `Line_Segment` and then applying the composition matrix to the `Line_Segment` object.



## 5) Object References

An object reference allows dynamic objects to contain references to other objects. Typically the class reference is instantiated directly as a reference.

```
void ref_example(Object* obj_to_ref, Object* obj_ptr)
{
    // Create reference to the object to reference (obj_to_ref)
    Reference ref_obj(obj_to_ref);

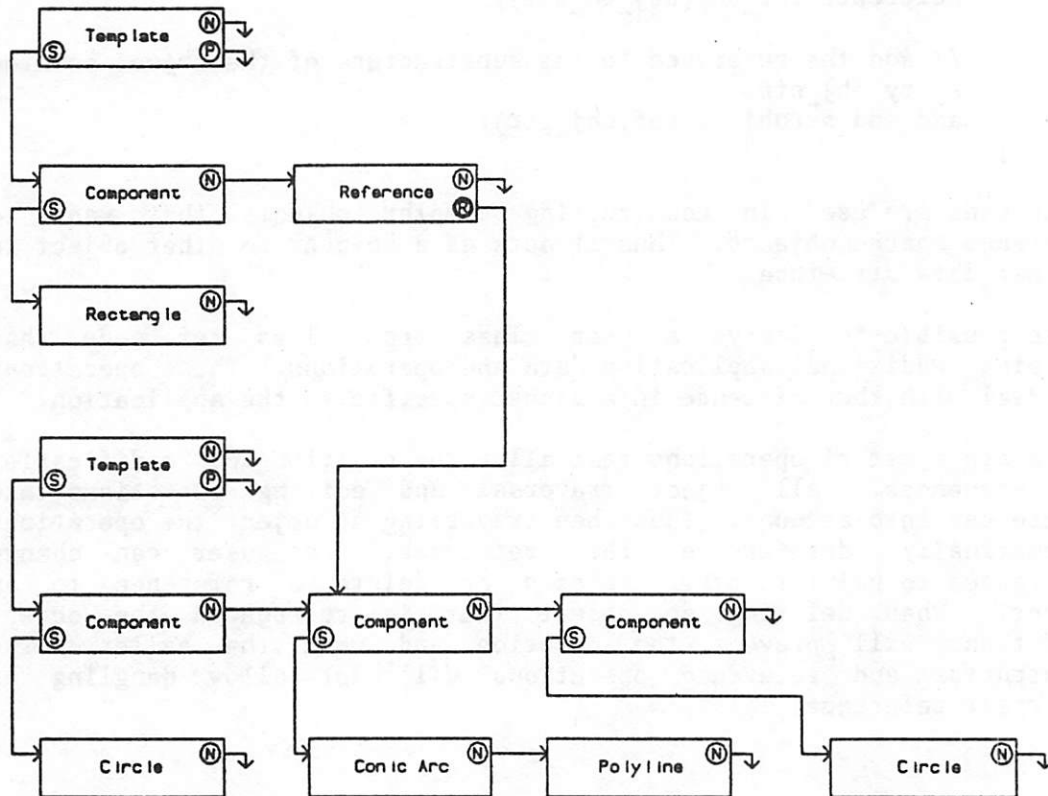
    // Add the reference to the substructure of the object pointed to
    // by obj_ptr.
    add_end_ss(obj_to_ref, obj_ptr);
}
```

References are used in constructing complex objects that want to reference shared objects. Thus it acts as a pointer to other object in another data structure.

It is possible to derive a user class from class reference that contains additional application data and operations. These operations may deal with the reference in a manner specific to the application.

There are a set of operations that allow the creation and modification of references. All object traversal and editing operations take references into account. Thus when traversing an object the operations automatically dereference the reference. The user can change references to point to other objects or delete a reference to an object. When deleting an object that is referenced the object operations will prevent the deletion and warn the caller. The constructor and reference operations will not allow dangling or incorrect references.

An example of a complex object using references is shown below:



## 6) Object Construction

The construction of complex objects is integral to the Object-Oriented Class Library. The concept of construction is: complex objects are constructed from simple or complex objects. Thus a dynamic hierarchical data structure can be easily built using the construction operations.

The construction operations are built around the basic classes and a set of construction rules and conventions. Normally the operations deal with these rules and conventions, keeping the details from the user, although in some cases the user is required to follow the rules and conventions or invalid objects can be built.

The following are the rules and conventions for construction:

An object is either simple (derived from a basic class) or complex (constructed from other objects).

Objects are constructed from simple or complex objects.

A list of objects is not considered a complex object.

Only the object operations are allowed to access the next pointer, the property pointer, the ancestor pointer, the reference pointer and the substructure pointer. There are a full set of operations (that can be extended) that allow access to these pointers. These pointers are private data.

If a user includes pointers in a data structure then it is required that the virtual function `wr_simple_obj()` be implemented.

The substructure is a list of simple or complex objects. The property list is a list of simple or complex objects.

A reference object can only point to an object that has been derived from class Leaf.

The ancestor pointer of a simple object must point to the object whose substructure or property list it is part of.

Except for diagnostics the user classes have no access to (or awareness of) the ancestor pointer.

There can be no circular references, otherwise the traversal operations would never complete.

## 7) Object Operations

The object operations are implemented as member operations and as functions outside the scope of any of the basic classes.

There are a set of operations referred to as the List Operations, these operations deal with class Element. Since a list is not an object, these operations are not referred to as object operations. These operations are used by the object operations and since all of the user classes have class Element in their hierarchy they can use these operations.

Thus there are List Operations & Object Operations. The Object Operations are divided into the following groups:

1. General Operations
2. Reference Operations
3. Operations on Substructure
4. Operations on Properties
5. Diagnostic Operations

The General Operations provide general traversal and editing operations for all objects. The Diagnostic Operations provide methods to inspect objects for debugging purposes.

### General Operations

common_ancestor	copy_obj
delete_list_obj	delete_obj
delete_obj_in_list	detach_from_obj
equal	next_object_type
in_obj	obj_deleted
object_written	read_obj
write_obj	

### Reference Operations

change\_reference  
external\_reference  
rm\_reference  
referenced\_external

### Substructure Operations

add_end_ss	add_end_ss_check
add_front_ss	add_front_ss_check
attach_ss_list	attach_ss_list_check

copy\_to\_end\_ss  
copy\_to\_front\_ss  
delete\_obj\_ss  
detach\_ss  
find\_ss\_obj  
object\_n  
rm\_object\_n

copy\_to\_end\_ss\_check  
copy\_to\_front\_ss\_check  
delete\_ss  
find\_list\_ss  
len\_of\_ss  
process\_ss  
ss\_of

#### Property Operations

add\_end\_property  
add\_front\_property  
attach\_property\_list  
copy\_to\_property  
delete\_property\_list  
detach\_property\_list  
find\_property\_obj  
property\_deleted

add\_end\_property\_check  
add\_front\_property\_check  
attach\_property\_list\_check  
copy\_to\_property\_check  
delete\_obj\_property  
find\_list\_property  
process\_property  
property\_list

#### Diagnostic Operations

ancestor\_of  
basic\_type

is\_object  
obj\_type

## 8) An Example of the Object-Oriented Class Library

The following is a set of user classes that implement a hierarchical graphical object called a Template. A Template consists of Components which are closed loop graphical objects. A Component consists of graphical primitives such as lines, curves or rectangles.

```
// TEMPLATE class

class Template : public node
{
    char name[16];                // Template name
    unsigned short num_components; // Number of components

public:

    Template(const char* str);
    char* name_is() {return name;};
    unsigned short components() {return num_components;};

    // Editing Operations for Components

    void insert_comp(Component *c);
    void delete_comp(Component *c);
    Component* select_comp(ucs_coor* point);

    // Drawing Operations

    void draw(dcs_clip clip_window, dcs_coor ulc);

    void rotate(double theta);
    void scale(double s);
    void scale(double sx, double sy);
    void translate(double dx, double dy);
};

// COMPONENT
// Component structure

class Component : public node
{
    unsigned short num_primitives; // Number of graphical primitives
    int fill_number;               // Fill number used

public:
```

```

Component() : (sizeof(Component),TEMPLATE)
    {num_primitives = 0;closed = FALSE;updated=FALSE; fill_number = 0;};
unsigned short primitives() {return num_primitives;};

// Editing Operations for Primitives

void insert_prim(Primitive *p);
void delete_prim(Primitive *p);
Primitive* select_prim(ucs_coor* point);
boolean point_is_in(ucs_coor* point);

// Rendering & Drawing Operations

void draw(dcs_clip clip_window,dcs_coor ulc);
};

// PRIMITIVE

class Primitive : public Leaf
{
    Primitive_Class primitive_class;
    Primitive_Types primitive_type;

public:

    virtual Segment* render(Transform_2D* t);
    Primitive(Primitive_Class classin,Primitive_Types typein):(sizeof(Primitive),
GEOMETRY) {primitive_class = classin; primitive_type = typein;};
    Primitive_Class prim_class() {return primitive_class;};
    Primitive_Types prim_type() {return primitive_type;};
};

// LINE

class Line : public Primitive
{
    ucs_coor pnt1;
    ucs_coor pnt2;
public:

    Line(Primitive_Class classin,ucs_coor p1,ucs_coor p2) : (classin,LINE)
        {pnt1 = p1;pnt2 = p2;};
    Segment* render(Transform_2D* t);
};

// RECTANGLE

class Rectangle : public Primitive

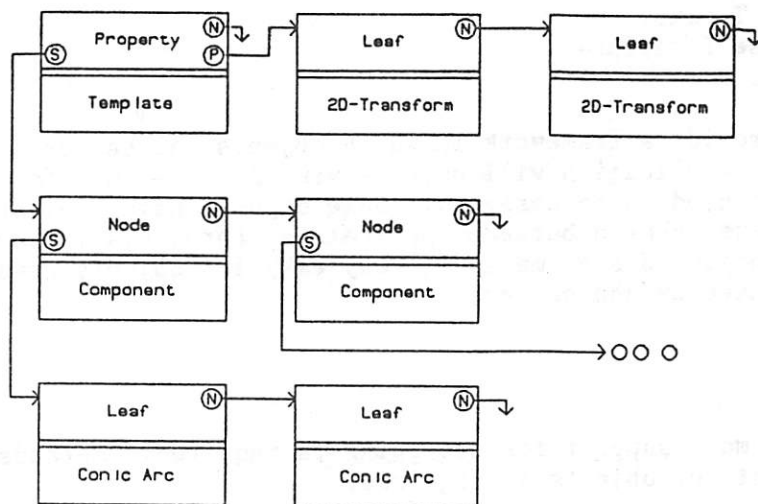
```

```

{
    ucs_coor ulc;
    ucs_coor lrc;
public:
    Rectangle(ucs_coor p1, ucs_coor p2) : (CLOSED, RECTANGLE) {ulc = p1; lrc = p2;};
    Segment* render(Transform_2D* t);
};

```

The following is a drawing of the Template object:



## 9) Applications

Currently this Class Library is being applied to applications that require objects to deal with:

- o An Ampex Real-Time Os Interface
- o X Windows Interface
- o User Interface Toolkit
- o Library Database Interface
- o And many others

The concept is to provide a framework in which objects can be designed, knowing that each application will have varying requirements for the type of objects that need to be designed. Some objects have universal appeal and thus are shared between applications (projects). Having this framework for object design makes it very easy for objects created by one group to be used by another group.

### **Extensions**

It was noticed that more support for debugging is required. Methods to print out the contents of objects is required.

Also for debugging support it would be nice to have an object walker that allows a programmer to walk through the object hierarchy and print the contents of various objects.

## 10) Conclusion

The concept behind the Basic Object-Oriented Class Library is to provide a methodology for the creation of dynamic persistent objects. These user created objects have specific applications. Some objects could have a universal appeal, such as OS interfaces, Graphics Interfaces, or general Database Applications. The goal is to provide a standard object-oriented approach to data structure design and allow the library to be easily extended. In the future it is desired to implement more I/O capability for printing the data structures as well as providing real debugging capability.

## References

1. K. Gorlen, Object-Oriented Program Support (OOPS) Reference Manual, National Institutes of Health, May 1986.
2. B. Stroustrup, The C++ Programming Language, Addison-Wesley, Reading Massachusetts, 1986.
3. B. Stroustrup, What is "Object-Oriented Programming"?, AT&T UNIX System V, C++ Translator, Technical Papers, 1987.

# Teaching C++

*Tsvi Bar-David*

MT3F-141

AT&T Bell Laboratories

200 Laurel Avenue

Middletown NJ 07748

## 1. What this paper is about

In this paper I discuss my experiences teaching C++ at AT&T Bell Laboratories over over the past year. The message that comes through loud and clear from the students is: "teach us object oriented programming so that we may make best use of C++." So the instructor's job is to teach object oriented programming and to map it into C++ facilities. The paper, to some extent, also reflects the collective experience of a number of C++ instructors in and around the AT&T R&D community. However responsibility for conclusions drawn from this experience rests solely upon myself.

## 2. Student Population

It is interesting to note how the student population has evolved over the past year. In late 1986 the population was composed of

- thrill seekers
- people whose office mate was using C++
- individuals, not groups

By mid 1987 the picture had changed radically. It was not untypical to see all the developers in a center sent to learn C++ *en masse* because center management decided to build their next product in C++. Demand for C++ classes has increased to the point where it is now necessary to schedule special offerings for requesting centers in addition to the regularly scheduled classes. Apropos, the desire to build products in C++ comes as much from programmers as it does from their managers.

## 3. Students Needs

The change in the student population from guru to (Bell Labs) average C programmer required changes in how C++ was going to be taught. This became clear from the questions that the students asked:

- Why am I learning this complicated language?
- What is it good for?
- How do I use it?

What follows is my answer to these questions. C++ is an object oriented programming language (and more). To get the best use out of C++, the student needs to learn the model it is based on. Therefore the student needs to understand concepts of object oriented programming. It helps even more if the student develops an enthusiasm for object oriented programming and for the fact that C++ supports this style of programming pretty well.

The data abstraction facility of C++ (*class*) alone encourages (though does not demand) spending more time on the design phase of a program. Students have a pretty fuzzy idea about what procedural design is all about, not to mention what is object oriented design. Therefore, it is useful for the student to learn principles of object oriented design and its relationship to procedural design.

#### 4. How I teach C++

Summarized here is how I teach C++.

- Ask the students what is a program and what is programming. Their usual answer is essentially procedure oriented programming.
- Now that the students have made their model of programming explicit, give them a new model, that of object oriented programming.
- Talk about the advantages and disadvantages of object oriented programming.
- Teach C++ syntax.
- Relate object oriented programming to C++ features.
- Give the students plenty of design and programming exercises in class.

The remainder of the paper is mostly what I teach about object oriented programming.

#### 5. Models of Programs and Programming

According to the procedural model, a program is a sequence of instructions or functions acting on passive data. There is no (language provided) connection between data and its natural operations, other than function invocation. In this context, the methodology of successive refinement can be stated as: what helping functions would it be nice to have in order to construct a function of interest?

To lay the groundwork for what follows, it is necessary to define type and object. A type is a set of values together with a set of operations on the value set. An object of a given type is a container for a value from the type's value set. Thus it consists of storage and value. C++ allows the programmer to define a type using the *class* mechanism. The type operations are called member functions. Here is a simple example that relates these concepts to C:

Value set: *int*  
Operations: *+, -, \*, /, %*  
Object: *int x = 1;*

The object model of a program, in contrast to the procedural model looks like this: A program is a community of objects of different types that accomplish the work of the program by by invoking each other's member functions. Each object — as a function of its type — determines its response, if any, to a member function. Member function names are resolved with respect to a symbol table private to the class. The name of the member function is independent of the implementation of

- the data of an object of given type
- the algorithm which implements the member function.

For some students it helps to relate these ideas to Smalltalk. Namely they can think of a member function invocation as sending a message to the object.

Although what is object oriented design is still the subject of vigorous debate, the C++ instructor has an obligation to say something intelligent about design. The following has met with appreciation on the part of both students and their managers:

- Define the *problem* domain
- Decompose the domain into a collection of objects of different types
- Specify each type
  - What are properties of a type?
  - Give the using programmer access to the properties via a *public interface* of member functions

- What are the dependencies between types?
- Implement each type
  - Successive refinement on types: What helping types would it be nice to have around to construct this type?
  - Select a data structure to represent object state.
  - Implement the member functions. As you structure the data so must you implement the algorithms. Here object oriented design includes procedural design, i.e. successive refinement on functions. For one can ask the question: what helpful (member or other) functions would it be nice to have around in order to write the given member function?
- Inheritance (factorization): Can I build a more complex type on top of simpler type(s)? That is, given a community of types, factor out common properties into a base class and derive the rest from it.
- Build a library of types and re-use them

## 6. Object Oriented Language Features

A working definition of an object oriented language is that it supports the following:

- Data abstraction
- Inheritance
- Dynamic Binding

## 7. Data Abstraction

The programming universe is a collection of different types (classes) of objects. Objects of a given type have operations that they may perform. This set of operations is called the type's public interface. The user of an object can manipulate it *only* by invoking its member functions. The user of an object has no direct access to the data that represents the object's state.

There are several advantages to this hiding of information: 1.) The owner of the type is free to re-implement it, usually for reasons of performance. A programmer using the type does not have to modify his/her code. 2.) Application developers are not tempted to commit the sin of permitting one object to depend upon the implementation of another.

Data abstraction is a way of thinking about programming. It can't be done efficiently in C, because of function call overhead, which is especially onerous given that object oriented programming encourages defining lots of small member functions. C++ has the class facility which, together with the inline facility, makes it easy and efficient to implement a data abstraction. However C++ does not force the programmer to do data abstraction. Indeed the programmer is free to define classes in which some of the data members are public. So it is important that data abstraction principles be an integral part of C++ training.

The students are taught to use class to define data abstractions. That is, all the data members are private. The public member functions constitute the interface. Private member functions are useful in encapsulating common algorithms that have to do with the implementation of the data abstraction.

To drive the point home, the students are given the example of an integer stack, and are then asked to implement the member functions *push()*, *pop()*, ... given two different representations of the stack data: contiguous array and linked list. The class definition looks like

```

class stack    {
    // implementation of data structure
public:        // public interface
    stack( int size); // create a stack
    void push( int x); // push x onto stack
    int pop();        // pop and return value
};

```

## 8. Inheritance

Inheritance is the ability to define a new type as a kind of existing type. Objects of derived type inherit from the base type all data and all member functions. In addition the derived type may define new member functions and additional data elements. The new type may override the base type definition of a member function.

>From a design perspective, the programmer decomposes the program into a set of objects of different types. S/he then looks for properties held in common by these types, *factors* the commonality into one or more base types, and then derives the rest of the types from the base types. The fundamental advantage here is re-use of code and data. Derived types are easier to implement the first time around since much of their functionality is already encapsulated in a base type. Furthermore, when a base type is re-implemented (correctly), *all* of its derived types inherit the improvements without having to be re-written.

C++ supports inheritance with the class derivation mechanism. More precisely, inheritance is public derivation. However, private derivation is *not* inheritance, since the programmer using a class derived privately has no access to the public member functions of the base class. Private derivation is not terribly different from defining the new class with an instance of the base class as a private data member.

>From an instructional perspective, one should teach public derivation as if it were the default (which it isn't — private derivation is the default). C++ syntax is unfortunate in this regard.

A nice example of inheritance is a collection of graphical objects. The base class *graph* factors out the common property that every graphical object has a position in the plane. Each derived class further specifies the geometry of its objects.

```

class graph {
    graph *next;
    int x, y; // position
public:
    // interface
};
class circle : public graph {
    int center, radius;
public:
    // interface
};
class box : public graph {
    int length, width;
public:
    // interface
};

```

## 9. Dynamic Binding

Before talking about dynamic binding, it is important to make sure that the students have a good picture of static binding, as practiced, say, by the C compiler. Here's our definition of static binding: All function names in a program are resolved (uniquely) with respect to one global

symbol table by the end of compilation. All function names are bound to a single body of code by the end of compilation. So, for example, in the program below, the identifier *printf* can refer to one and only one function regardless of its arguments or the context in which it is used.

```
main()
{
    int n = 3;
    printf( "%d0, n);
}
```

## 10. Dynamic Binding

Dynamic Binding is the ability of an object — as a function of its type — to determine at runtime what function it will call in response to the invocation of a member function. C++ lets the implementor of a class choose the binding. The default is static binding.

Dynamic binding is particularly useful for container classes, like linked lists, which can contain different types of objects. The designer of the container class places some minimal requirement on the objects to be contained, for example that they each must have a *print()* member function for displaying themselves to the terminal. In this case the list class implementor writes its *print()* member function once, and it will work properly for all types of objects that adhere to the requirement. This will be true even for types that did not exist at the time that the list class was implemented! This is extensibility.

In C++ it takes a fair amount of work to set up to do dynamic binding, so it is good to give the students an example that they can take home with them. The basic idea of the example is a list class for heterogeneous objects. The list can display itself. Each object belongs to a class with a *display()* member function. The trick is to publicly derive all of these classes from a base class in which *display()* is declared *virtual*.

The example list class is used for the internal representation of a picture manipulated by a graphics editor. It contains objects of classes derived from our old friend *graph*, such as *line*, *box*, *circle*.

```
class graph    {
public:
    ...
    virtual void display();
};
class GraphList : public List {
public:
    ...
    void display()
    {
        for( graph *p = head ; p ; p = p->next)
            p->display();
    }
};
class rosewindow : public graph {
public:
    void display()
    {
        ...
    }
};
```

Here is what the list display function might look like in C. Each object has an explicit type tag which contains the address of a function switch table. Each object finds its display function at a

common index in its switch table. In contrast, in the C++ implementation, there is no need for a type tag; it is the class to which the object belongs. The disadvantage of the C implementation is managing an explicit type tag, and all the tedious error-prone book-keeping involved in setting up the switch tables.

```
#define DISPLAY 1
display( list)
{
    graphlist *p;
    for( *p = head ; p ; p = p->next)
        (*p->type[DISPLAY])( p);
}
```

## 11. Citizenship

The concept of object citizenship is a useful teaching tool that unites and clarifies certain features of C++. Citizenship has to do with how the language lets programmer manipulate values with respect to: assignment, function invocation — the capability of being an argument or return value, aggregation — array, structure, and derivation. First-class citizenship is defined as how builtin types are manipulated. C++ user-defined objects are first-class. Oddly, builtins are in some sense not first-class in that they cannot serve as a base for derivation. C++ classes are not first class citizens, as they are in Smalltalk. However they may have a representation of state, via *static* data members within the class. A class also has a (limited) public interface consisting of *new()* and constructors.

# Modelling Graphical Data with C++

Al Conrad

University of California  
at Santa Cruz  
November 10, 1987

Many applications in graphics (drawing programs, animation editors, renderers, etc.) are written in C. These applications typically employ an internal structure for modelling the graphical scene which consists of a linked list of shapes. Each shape on the list is a *struct* with a field indicating the shape's type (circle, polygon, spline, etc.) and a data field. The data field is typically a C union containing a pointer to a *struct* with the shape specific data. Operations like **draw**, **resize**, or **rotate**, are then implemented as large case statements which switch on the shape's type.

Of course, C++ provides a more sophisticated mechanism for dealing with a polymorphic structure like a list of shapes. Circles, polygons, and splines become specific derived classes of the base class **shape**.<sup>1</sup> The monolithic case statements within **draw**, **resize**, and **rotate** can be broken out into virtual functions so that all of the operations for a given shape can be grouped in cohesive modules.

The first interesting question that rises out of designing a C++ class hierarchy for collections of shapes is: *What information belongs in the base class, shape, versus what information must be factored out into the individual derived classes, circle, rectangle, etc.?* While, **radius**, for example, is clearly a shape attribute that is specific to **circle**, perhaps **x**, **y** location can be thought of as an attribute common to all shapes and thus maintained in the base class. This has the immediate advantage of allowing shape methods like **translate** to be implemented once in the base class, rather than having to provide a separate **translate** for each circle, rectangle, etc. In practice, however, this causes problems for shapes which are typically represented as point lists and therefore have no canonical *center* (e.g., polygons, splines, etc.).

For this reason, in implementing **tinker**,<sup>2</sup> we slipped into the habit of keeping all shape attributes in the individual shape classes with only the list linkage information surviving in the base class. Following are the class declarations for **picture**, **shape**, and **line**:

<pre>class picture {     shape* head;  public:     void draw();     void rotate(double,int,int);     void scale(double,int,int);     void move(int,int);     void set_thickness(int);      void operator=(shape*);     void operator+=(shape*);     void operator--(shape*);</pre>	<pre>class shape {     friend class picture;     shape* next;  public:     virtual void draw();     virtual void rotate(double,int,int);     virtual void scale(double,int,int);     virtual void move(int,int);     virtual void set_thickness(int);     virtual int in_rect(int,int,int,int)</pre>	<pre>class line : public shape {     int x1, y1;     int x2, y2;     int thickness;  public:     void draw();     void rotate(double,int,int);     void scale(double,int,int);     void move(int,int);     void set_thickness(int);     virtual int in_rect(int,int,int,int)</pre>
--	--	--

In hindsight, style attributes like line thickness could have easily been accommodated in the base class. Moreover, location information could also have been maintained in the base class if a more general approach had been used for the 2D graphics. Specifically, if each shape were specified within its own coordinate system, then a *current transformation matrix* could be kept in the base class. Although this is a common approach in 3D computer graphics systems,<sup>3</sup> performance considerations and historical constraints have locked most window managers into the rasterop, integer arithmetic, absolute coordinate system world. X is no exception, and thus for this implementation of **tinker** our partitioning was justified. The NeWS<sup>4</sup> window system, however, is an exception and in **tinker**'s successor (**post-tinker**), being written for the NeWS environment, the *current transformation matrix* approach can be used and thus all

transformation information will be maintained in the base class.

Many of the operations performed on a list of shapes required for a drawing program like *tinker* are reminiscent of set theoretic operations. In particular, adding a shape to the picture is like adding an element to a set; combining two pictures is analogous to forming the union of two sets. With C++ it is then tempting to utilize the operator over-loading mechanisms so that we can write `p1 + p2` for two pictures `p1` and `p2`, instead of `union( p1, p2 )`. Similarly, we can add a shape `s` to a picture `p` by writing `p = p + s` (or better yet, since we are C programmers: `p += s`).

In *tinker* we employed these overloaded operators and found that they improved the code readability. For example, the code to form a sub-picture from all shapes falling within a given rectangle looks something like:

```
s = p.head;
do
    if ( s->in_rect( x1, x2, y1, y2 )
        subpic += s;
    while ( s = s->next );
```

Buried in `in_rect()` is the Cohen-Sutherland<sup>5</sup> algorithm for detecting an intersection with a rectangle. A nicer approach planned for *post-tinker* is to overload the `^` operator to mean set intersection, so that the above code could be written:

```
subpic = s ^ r;
```

Few mechanical difficulties were encountered in accessing the X libraries from C++ (with the exception that `Xlib.h` had to be modified to accomodate function prototyping). Functionally, however, the limitations imposed on X by the rasterop model<sup>6</sup> made it difficult to fully explore the opportunities afforded by C++ to utilize object-oriented methods for implementing standard graphics mechanisms (e.g., hierarchical models with instance transformation matrices,<sup>3</sup> grouping and un-grouping,<sup>7</sup> and accomodating images<sup>8</sup>). With NeWS,<sup>4</sup> the target environment for *tinker*'s successor (*post-tinker*), the problem is reversed. The foundation for NeWS is PostScript<sup>9</sup> which not only provides the graphics functionality missing in X, but is a general purpose language in its own right with object oriented extensions. An immediate question for *post-tinker*: *Where does the shape class hierarchy belong: in the C++ client code or in the PostScript sent to the NeWS server?*

## References

1. Bjarne Stroustrup, *The C++ Programming Language*, p. 216, Addison-Wesley, Murray Hill, NJ, 1986.
2. Ti Kan, *Tinker: An Interactive, Object-oriented Drawing Program*, University of California at Santa Cruz, Santa Cruz, CA, 1987.
3. J. D. Foley, A. Van Dam, *Fundamentals of Interactive Computer Graphics*, p. 342, Addison-Wesley, 1982.
4. *NeWS Technical Overview (800-1498-05)*, Sun Microsystems, Inc., Mountain View, CA, 1987.
5. J. D. Foley, A. Van Dam, *Fundamentals of Interactive Computer Graphics*, p. 146, Addison-Wesley, 1982.
6. William M. Newman, Robert F. Sproul, *Principles of Interactive Computer Graphics*, p. 262, McGraw-Hill, 1979.
7. *MacDraw User's Manual*, pp. 19-20,39.
8. *NeWS Manual (800-1632-10)*, p. 127, Sun Microsystems, Inc., Mountain View, CA, 1987.
9. Adobe Systems, Inc., *PostScript Language Reference Manual*, Addison-Wesley, 1985.

# Integrated Class Structures for Image Pattern Recognition and Computer Graphics

James M. Coggins  
Computer Science Department  
University of North Carolina  
Chapel Hill, NC 27599-3175

## Introduction

Research efforts in image pattern recognition and computer graphics face two kinds of software problems. First is the intrinsic complexity of the algorithms developed in the course of the research. Second is the design and construction of the researcher's software toolbox including fundamental operations, standards for data storage and communication, and interfaces to rapidly changing sets of display and interaction devices. The complexity problem is intrinsic to the subject matter and objectives, and it is the proper domain of the researcher. The second kind of problem is incidental to the research objectives and can be addressed by adoption of modern software development tools and disciplines, including object-oriented design for code and hypertext structures for documentation.

This paper describes several techniques we have developed while designing an integrated object-oriented software toolbox for image pattern recognition and interactive computer graphics research. Design criteria for the system include (1) pervasive integration of constructs, (2) maximum flexibility for researchers using the system, (3) minimum user effort to invoke the facilities, and (4) purity of the object-oriented design. We will describe in this paper techniques we have developed for managing massive data structures, providing type-independence at the user level, encapsulating device dependencies, processes, and class interfaces, and decomposing the required software system while maintaining integration of concepts.

## Managing Massive Data Structures

The kinds of structures we manipulate (images, pattern matrices, graphical models) often have large or *very* large memory requirements. We do not want to reallocate, copy, and destroy these large structures in each function call and function value return. Instead, we implement large structures using a **header object** that stores descriptive data about the object along with a pointer to a **storage object** that contains the data. The header class `image`, for example, contains the image size and shape, its

disk file name, history, and other data, plus a pointer to an object of class `buffer` which contains the pixel data for the image. Class `image` understands messages that will be forwarded to the `buffer` object for processing. For example, `image image::operator+=(image&)` must be defined, but its action is simply to invoke the `buffer` method `void buffer::operator+=(buffer*)`. Objects of class `image` are small, so they may be copied, allocated, and destroyed as needed with negligible performance penalty. We must be careful, however, in the constructors and destructor of `image` to avoid repeated allocation, copying and deallocation of large `buffer` objects. This means that the `image::image(image&)` and `image::operator=(image&)` messages must *copy* the buffer pointer of the source image and not allocate a new buffer. In order to prevent the destructor `~image()` from deallocating the buffer while it is still being used, we place a reference count in the `buffer` object and delete the `buffer` only if the reference count is zero after decrementing. This mechanism also prevents the large `buffer` objects from being left in the heap as garbage, which would soon result in exhaustion of virtual memory. We are planning to use the header class/storage class implementation for all of our large structures such as pattern matrices and some graphical models.

### Type independence

The separation of header and storage classes also makes possible type independence at the user level. Storage types for images include byte, color (4 bytes for RGBs), integer (short), real (float), and complex (a pair of floats). Objects of class `image` are manipulated as desired, independent of the storage type. Messages involving the pixel data are passed on to the image's `buffer` object. The `buffer` class contains virtual function declarations for the suite of operations affecting the pixel data. The real work is performed in subclasses of `buffer` that are type-specific: `byte_buffer`, `color_buffer`, `int_buffer`, `real_buffer`, and `complex_buffer`. Necessary coercions are provided between the buffer subclasses. When more than one coercion is possible (complex-to-real can be performed by real part, imaginary part, magnitude, or phase), a default is assigned and an optional parameter can be used to override the default.

Since the `image` objects are, in effect, annotated pointers to buffers, `image` objects can be manipulated naturally in expressions such as `result=im1+im2*2.0;` without the distractions of creating and dereferencing pointers.

The overhead of this extra level of indirectness is negligible because by assumption we are manipulating large objects. Since we get both type-independent manipulations and a reasonable method for managing the large memory requirements, we do not begrudge the overhead cost.

## Encapsulating device dependencies

A serious problem in a high-technology lab such as ours is keeping the software base current and consistent with the available hardware capabilities. We have experienced a phenomenon we call "hardware indigestion" in which we have difficulty incorporating new hardware into existing projects because of incompatibilities between the various devices and the device-specific nature of the controlling software supplied by the vendors. Of these issues, the software incompatibility is the more serious problem. Advances in graphics and imaging devices have usually involved speed and resolution enhancements, not entirely new functionality. Accessing that functionality is difficult because the vendor's software involves intricate code that is optimized in some sense for the device's capabilities and that is not amenable to incorporation into a uniform interface.

We have worked on three kinds of device encapsulations. Disk file manipulations are supported by a class `diskfile` that handles basic disk operations such as open, close, read, write, and seek. These operations are then invoked by derived classes that "know" the structures of particular kinds of disk files such as `imagefile`, `polyfile`, and `patternfile`. New kinds of file structures can be added as derived classes of `diskfile` without changing any of the existing code.

Another kind of device encapsulation we have developed involves analog input devices such as knobs, joysticks, and sliders. The key to the design of these classes was recognizing that the only differences between them from the system's viewpoint are the name of the device handler and the number of bytes expected from the device in a single read operation. Operations provided by the abstract superclass include `poll` to force a read of the device and `int rawdata(int)` to obtain one of the values provided by the A/D converter. A uniform user interface is provided by adopting the convention that the device-specific classes convert the integer raw data value to a double between 0.0 and 1.0. Now the roles of the devices can be interchanged by simply declaring the device object to be of a different device subclass. Device-specific interfaces are also available; a 2-D joystick can return a point, and a 3-D velocity joystick can return a vector.

The third kind of device encapsulation involves display devices. We decided to make the unit of encapsulation be the viewing surface, so on window-oriented systems, the display object created is a window. Thus, several display objects may be active at once on a device. Several basic capabilities are defined in the abstract superclass including clearing the display, drawing lines, writing text, rendering polygons, and displaying

images. We are still debating how the enhanced capabilities of some devices may be made available within this framework, especially when the architecture of the device requires the data to be prepared differently for display processing.

### **Process encapsulations**

Since our research involves development of new algorithms for imaging and graphics problems, encapsulation of these frequently-changing processes is essential to our research software environment. We have used a technique called *process encapsulation* to simplify the use and invocation of the processes based on the dictum "Encapsulate most deeply that which is most likely to change." Conceptually, a process encapsulation creates an object that we call an *enzyme* or a *catalytic object*, whose purpose is to mediate interactions among other objects. In a process encapsulation, a class structure is defined for the process type, a renderer or a classifier, for example, that specifies the minimum functionality and parameters of such a process. Then specific algorithms are defined as derived classes with their own parameters as required. To use the process, we create an object of the desired subclass, connect it to other objects and supply the parameters it needs, and send it a "begin" message. The input objects and parameter values can be supplied in three ways: by arguments to the constructor, by assignment in separate messages to the object, and by arguments to the "begin" message. This design allows the user to customize the process in a separate code segment from that where the process is invoked, leading to very clean code for the basic algorithm that invokes the process. The inheritance of fundamental operations and structures from the process' base class contributes to rapid development and evaluation of algorithm modifications and parameters.

### **Class Interface Encapsulations**

The definition of standard process interfaces is facilitated by classes that store intermediate results in a standardized form (or a set of agreed-upon forms). Some details of this design innovation are still under development, but a typical example occurs in a graphics pipeline where various kinds of object models must be converted into "rendering primitives" that the display devices understand and can process. By adopting a standard set of rendering primitives, developers of process encapsulations for renderers and developers of display device encapsulations are insulated from each other's internal data structures and processing requirements. Development of renderers can proceed in a device-independent fashion and augmentations to the set of rendering primitives are explicitly noted and handled by all device encapsulations. By standardizing interface classes, most of our graphics research efforts can begin to share code. We find that research that is advancing the state

of the art sometimes still must diverge from the standards either in order to optimize performance or in order to explore new paradigms that are beyond the state of the art. An example of the former case is real-time interactive graphics using customized parallel architectures requiring a different structure in the graphics pipeline. An example of the latter case is research in texture mapping in which the graphics pipeline is modified to accomodate an entirely new kind of rendering.

In order to accomodate these research efforts, our graphics class structures are designed for ease of use by *system developers* and has been criticized for being less than optimal for users. This is an explicit design tradeoff that we have accepted in order to provide flexibility and control at the expense of some ease of use and fidelity to user-level conceptual structures.

### Separation of Concerns

The interface classes described above are used to implement a separation of concerns that has guided the design of our basic class structures. We will illustrate its effect with an example from our image processing library.

In our first implementation of class `image`, we included messages such as `load`, `save`, and `display`. The resulting structure had several problems. First, putting everything into `image` made the code too large. Second, the code for `image` had an unpleasing asymmetry. The code for the `load`, `save`, and `display` messages overwhelmed the code for the numerous image processing messages, most of which were less than ten lines each. Third, the intricate code we worked out for handling disk I/O was unusable by any other classes, and the `display` operation was useless on any but the device we defined it for.

The next incarnation of `image` separated the concerns of storage, processing, and interaction devices into different classes. Storage was handled by a class, `diskfile`, that encapsulates all low-level disk operations but without any knowledge of the semantics of the file being manipulated. A subclass `imagefile` directs the decoding and interpretation of the disk data. The `image` class retains the processing operations. Display of images was moved out to a `display_device` class with subclasses for the various devices available in our lab. Thus, an "image" became a "rendering primitive" that all display devices are expected to process in some reasonable manner.

The principle of separation of concerns is primarily an implementation principle that helps to provide the control and flexibility that we need in our research environment, but it sometimes works against the kind of

user-level ease-of-use and fidelity to conceptual structures that is a hallmark of Smalltalk. We are still investigating whether and how a user-level class structure might be imposed atop our implementation structures without redesign for each alternative implementation of a graphics or imaging pipeline.

### **Conclusion: Is C++ Really the Right Tool?**

Object-Oriented Programming is a code packaging discipline that imposes a reasonable structure on large bodies of code, with additional benefits of code sharing within each class hierarchies and effective conceptual metaphors for talking and thinking about programs. Object-oriented programming provides just the kind of discipline and structure that we need as the size and complexity of our software base increases beyond a level where a single person can maintain, control, and understand it. Since we are an established UNIX environment, a C derivative makes sense in view of our large installed base of C code and our need for implementation control in order to support real-time operations and efficient handling of large storage structures. These properties of our environment and objectives make C++ a reasonable language for our software development efforts.

We eagerly await the development of a symbolic debugger for C++ and some relief to the problem of proliferating header files and the compile-time overhead they require. A precompilation of header files into an "environment file" similar to those available for DEC's VMS Pascal would be, for us, an ideal solution to the problem.

Our software development efforts are proceeding on two levels. Low-level encapsulations of basic data structures are being debated and sometimes shared among implementers. High-level architectures for large structures such as graphics and imaging pipelines are also being designed and debated. We believe that the largest benefit will be obtained from the high-level architectures and the standards decided at that level, but the low-level encapsulations are more immediately useful to implementers who already have their own versions of the major processes in the laboratory. Whether anticipated revisions to C++ such as parameterized types impact the effectiveness or generality of these design activities remains to be seen.

### **Acknowledgements**

This research was supported in part by ONR contract N00014-86-K-0680. Students Muru Palaniappan, John Rohlf, and Brice Tebbs have made significant contributions to the design and implementation of the integrated class structures. About a dozen other faculty and student researchers in our department's Graphics and Imaging Cluster have provided valuable advice and criticism.

# Using C++ to Develop a WYSIWYG Hypertext Toolkit

Jim Waldo  
Apollo Computer Inc.  
330 Billerica Rd.  
Chelmsford, Ma. 01824  
decvax!apollo!waldo

This paper is essentially a case study in using C++ to develop a toolkit for text applications. I will begin by giving an overview of the toolkit itself, focusing on the ways in which it differs from standard methods of dealing with text. I will then discuss how the toolkit utilizes various features of the C++ language, and how certain features of the language allow the actual implementation of the toolkit to more closely resemble the optimal system envisioned during the design process. I will conclude by discussing some of the dark linings of the silver cloud of C++ and discuss some of the methods we are using to attempt to overcome these problems.

## The Text Management Library

The toolkit we are developing, known internally as the Text Management Library (TML), is meant to provide all the functionality needed to produce a broad range of text applications, from viewers to simple plaintext editors to structured document or program editors. The requirements specified that it should handle multiple fonts and character sets (both one and multiple byte per character), allow various forms of formatting control, support hypertext facilities for multiple paths through a document, and allow the presentation and manipulation of such text in a true WYSIWYG fashion at interactive speeds. Further, since TML is a toolkit, we realized that it would be used in ways that we could not foresee. This added the requirement that the library be extensible by the user in as general a fashion as possible.

We quickly realized that a library that would meet all of these requirements could not be based on the usual model of text that treats text as a linear stream of codes with escape sequences to mark off information about the text. Instead, we decided to base the library on a model of text that makes a clear distinction between the characters that make up the text, the structure that organizes the text, and other information about the text. We refer to this model of text as the *hierarchical object* model, for reasons that will become apparent below. For a more complete description of this model of text, see [4].

The first step in arriving at this model for text was to take the notion of *structure* in text seriously. Text is rarely a simple stream of characters. A document such as this paper is, at bottom, a stream of characters; but those characters are grouped together to form sentences, that are in turn grouped together to form paragraphs, that in turn are grouped together to form the final object that is the paper. More complex documents might have structural elements such as sections (made up of paragraphs), chapters (made up of sections), and volumes (made up of chapters). Source files are also made up of a base of characters, but these characters are organized in a different fashion — into lines that make up blocks that make up functions, etc.

To reflect this kind of inherent structure, we decided to represent text using a generalized threaded graph structure in which the terminal nodes of the graph refer to linearly ordered streams of character codes. Non-terminal nodes serve the purpose of imposing structural order on the text but do not refer directly to character codes. It is natural to consider each node in the graph as an identifier of a text object that consists of the sub-graph dominated by the node.

This notion of structure is not in itself new; it is similar to that found in most systems that claim to support a hypertext organization [3] and has much in common with the organizational principles used in

various "structure-based" editors [2]. Indeed, if one follows the simple definition of "hypertext" as "non-sequential writing"<sup>1</sup> hierarchical text objects are examples of hypertext. The structure of a hierarchical text object, however, plays a far more central role than is usual in other text systems based on the hypertext notion. On our model, the basic ordering of the characters in a hierarchical text object is determined by the structure, which may also be used for purposes of referring to other sections of the text object or to other text objects.

The treatment of text as a hierarchical object differs from that in structure-based editors in two ways. The first is that the hierarchical object model is more general. No interpretation is inherent in the various structures; all that is assumed is that the text is to be structured in some way. The second difference is that the structure in the hierarchical object model is taken to be an essential part of the text object. The structure graph is present both in memory and on disk. The text is never stored in a purely linear form.

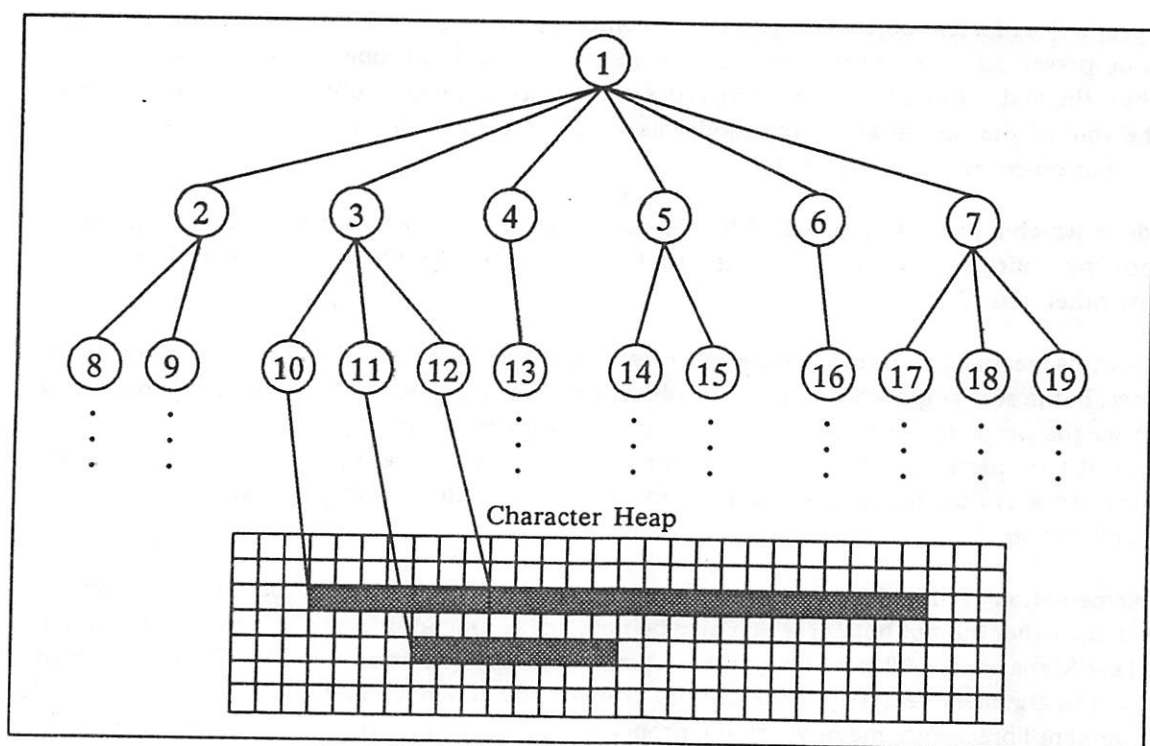


Figure 1. Partial Diagram of a Generic Hierarchically Structured Text Object

Part of the information carried by such a text object is inherent in the organization of the graph. For example, it is the configuration of the objects within the graph that determines the linear ordering of the characters that are pointed to by the terminal constituents of the graph. Character codes themselves never appear in the graph. Instead, the characters that make up the object are stored in a single heap, and terminal nodes in the text structure point to areas in the heap that contain the characters dominated by that node. Note that the global ordering of characters within this heap need not have any relationship to the ordering of the characters in the text object. Linearity is maintained within the sub-areas of the heap pointed to by a particular terminal node, but no ordering is maintained outside of such sub-areas. This character heap can be thought of as a sort of primordial character goo, in which order is maintained only in very local areas. It is the hierarchical object structure of the text graph that provides global form and structure to the text object.

It should also be noted that the character goo is free of any character codes that attempt to convey information about the text. All codes within the character goo are interpreted as characters that will appear in the final rendition of the document, and none of them serve as directives concerning the placement of the characters that follow. Figure 1 shows a partial diagram of an example of a generic structure of this sort (generic because we have supplied no interpretation for the objects in the structure).

Since the toolkit must be sufficiently general for applications working with any sort of text, the Text Management Library does not presuppose any interpretation of the structure-inducing nodes other than that which arises out of their position in the threaded graph. The objects in the text graph by themselves impose order in that some are parents or siblings to others, but the effect of this imposed order is not inherent in the objects themselves. The objects that make up the nodes of the structure graph and that impose order on the characters in this model are themselves featureless containers relating only among themselves and whose influence can only be seen indirectly through the properties attached to them.

It is the properties of a text object that give interpretation to the nodes of the graph. Determining how the text is to be presented is one of the functions of the properties. But properties are not confined to determining how the text is to be formatted. Properties can also be applied to objects to convey information about the role of the object within the model itself or to convey information about how the text dominated by that object is to be rendered.

The node to which a property is attached is used to determine the scope of the property. A property is activated for the entire text object determined by the node in the graph on which it resides and is inactive in any other text object.

Because of this mechanism, properties are inherited. Once a property is active it remains in force for all of the objects in the sub-graph whose root is the object having that property. The only exception to this is in cases where the property is explicitly over-ridden by a property of an object that occurs as part of that sub-graph. If a property is over-ridden, it remains inactive only for the sub-graph whose root is the node having the property that did the over-ride. On exit of that node the property that was over-ridden is once again made active.

Our implementation of properties treats them as a pair of functions with associated argument vectors. The two functions, either but not both of which may be null, act as functions from the argument vectors to states of the Text Management Library. One of these functions is the activation function. When combined with the first of the argument vectors provided in the property this function changes the property state of the text management library into one in which the property is active. The second function has the responsibility of returning the state of the Text Management Library to what it was prior to the execution of the activation function. Calling this function with the appropriate arguments has the effect of making the property inactive.

The toolkit supports two separate classes of properties. The first, which are called system properties, are supplied as part of the toolkit. Knowing full well, however, that we will not be able to predict all possible uses of the toolkit, we also provide support for a separate set of user-defined properties. Such properties have the same form as the system properties; however, the function table in which the entry and exit functions for such properties reside is available to the application. Thus the application developer is able to write his or her own entry and exit functions that may be loaded into this table, create properties that access these functions, and thus add those properties to the set defined in the toolkit.

Perhaps the most obvious use for properties is to specify formatting information for the text object. Properties are used to determine the size of the area in which the text is to be formatted, the font in which the text is to be presented, and the setting of tab tables.

Properties are also used to store information about the way the text is to be rendered. Setting the foreground and background color to be used in painting the characters, for example, can be done by associating a property with a structure object.

Finally, properties can be used to convey information about the text model itself. Objects can be named via a property, and relationships between parts of the text object may be reflected with properties. Thus, even though there is no object that is inherently a paragraph in the toolkit, a structured document editor being developed using the toolkit could create a property "being named a paragraph" and associate that property with certain objects. Having established such a property, the application could further require that certain other properties are always associated with objects having that property (such as being formatted in a certain way), or that only objects with certain properties (such as the property of being named a sentence) could be children of an object with that property.

The distinction made above between formatting properties, rendering properties, and modeling properties is hardly accidental; indeed, this three-fold distinction in the properties reflects a basic division of labor in the toolkit. The toolkit contains separate modules for manipulating the text model, for formatting that model, and for rendering a formatted version of that model.

This division of labor is made possible by the decision to reflect only the structure inherent in the text itself in the hierarchical text object and not the structure placed on that text by the formatting. By making such a separation, the text object may be considered to be format-independent. Thus, changing the format of such a text object does not change the basic underlying structure of the object itself. The distinction between the formatted version of a text object and the rendering of that object is a way of making the text object output-device independent. The rendering component takes as input a formatted text object, and translates that formatted object into the particular visual image appropriate for the current output device. If the text is being output to a printer, the rendering of the formatted object will look as much like the ideal formatted object as the resolution of the printer and the available fonts allows. However, if the output device is a screen, a number of options are available to the application. The screen (or the current window on the screen) may be treated as a viewport on to the formatted text, thus showing only a fraction of the ideal output, or the text object may be reformatted on the fly to fit into the output area available on the screen.

The division of labor is reflected not only in the operations performed by the various modules but by the basic entities on which those modules operate. The modeling component is fundamentally concerned with the hierarchical text object and operates on the graph structure of the objects, the properties associated with those objects, and the character heap. The formatting component translates this hierarchical text object into an abstract two-dimensional representation. The basic objects in this representation are the conceptual page and the conceptual line. A conceptual page has a width and a depth; it is on such conceptual pages that the text is laid out according to the formatting information contained in the properties present in the text object. Conceptual lines are used to fill these conceptual pages. Conceptual line objects also have a width and a height and are associated with a part of the model that is contained within the line. It is the job of the rendering component to translate these conceptual pages into images, either printed or painted on a screen. To do this, the rendering component maps the conceptual page into a viewport, which may be larger, smaller, or the same size as the conceptual page.

Interactions may take place on any of the three levels of the toolkit. Changes to the model may alter the structure graph, add to that graph and, perhaps, the underlying character heap, or alter the properties associated with the objects in the model. Such changes will be permanent (unless undone). If the change made alters the appearance of any text which is currently visible in a viewport, those changes will be

immediately reflected in the view of the text object . Temporary changes to the formatting or rendering of the text object may also be accomplished by interactions with those components; such changes will not be saved in the object and hence are local to the particular editing or viewing session in which they are made.

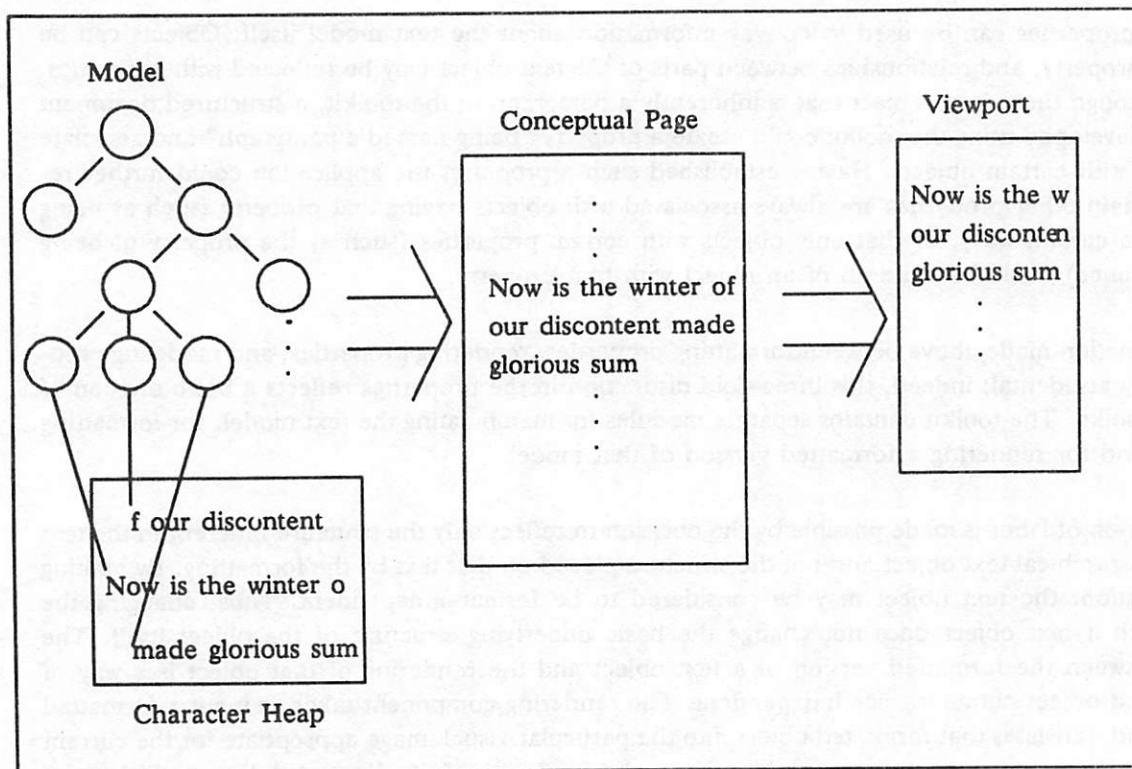


Figure 2 Mapping from Model to Conceptual Page (Formatter) to Viewport (Renderer)

The library has had a rather interesting history that has included two completely separate implementations. The first implementation was done using an extended version of Pascal as the base language. Given the object-oriented design of the library, the matching of language and system was not a happy one. While we were able to produce most of the functionality required in the library, we were unable to allow much in the way of user extensibility of the library beyond the introduction of user defined properties. It was then decided that the library should be reimplemented using C++. It is to this implementation that we will now turn.

## Using C++ to Implement TML

The first major change we noticed when we began redesigning TML for implementation in C++ was that the design activity felt much less like standard software design than it did like traditional metaphysics. Rather than concerning ourselves up front with data structures and algorithms, we spent our time in delineating the objects that would make up the TML world, what the relationships between those objects was to be, and what operations would be allowed on those objects.

We began by identifying the largest objects in each of the components. For the modeling component we identified the text model, which would be a graph with its associated nodes, properties, and characters. The model object also contains the object workspace for all of the entities associated with a particular maximal text object, including the heaps from which all of the other entities would be allocated.

The largest object in the formatting component is the format, which can be thought of as a connected set of conceptual pages. We added the ability to associate properties with a format; such properties would over-ride the properties set in a model being formatted within a particular format, thus allowing the building of applications that could display text in a unified way no matter what the properties of the displayed text object happened to be. Unlike the model, the format is a transient entity, created when the text model is to be displayed or printed. Thus the format object does not need to establish its own workspace, but simply allocates the objects that are parts of the format from general free space.

The largest object in the rendering component is the viewport, which roughly corresponds to an output device such as a window on a display or a printer page. Again, we added the ability to add properties to a viewport; these would override any properties in a format or model being displayed in that viewport, thus giving an application a third level of control over the presentation of the text. Like the format, the viewport is a transient object, and thus does not require its own workspace for the allocation of the entities that make up the viewport.

Only one sort of relationship holds among these highest level objects, that of being attached. Models may be attached to formats, and formats may be attached to viewports. The semantics behind this relationship is straightforward—the format a model is attached to is the container for the two-dimensional representation of the text in the model, and the viewport a format is attached to is the canvas upon which the formatted text is displayed. A model may be attached to more than one format at any given time, and a format may be attached to more than one viewport at a given time. This allows multiple layouts of the same text to coexist, as well as multiple views of a layout or set of layouts.

Each of these three main objects can be thought of as containing their own universe of objects. While the objects may refer to other objects outside their universe, the main interactions are all within the objects of a particular component.

The richest universe is found in the model. The main objects used here are the node object, the property object, and the property argument object. The simplest of these is the property object, which simply identifies a sort of property such as a font property, a node identifier property, or a wrap mode property. Properties are not fully instantiated unless they are associated with an argument vector. These argument vectors map the property type onto a particular instance of that property, such as the property of being in font times-roman 12 pt. or being a node identified as a paragraph or being a word-wrap property.

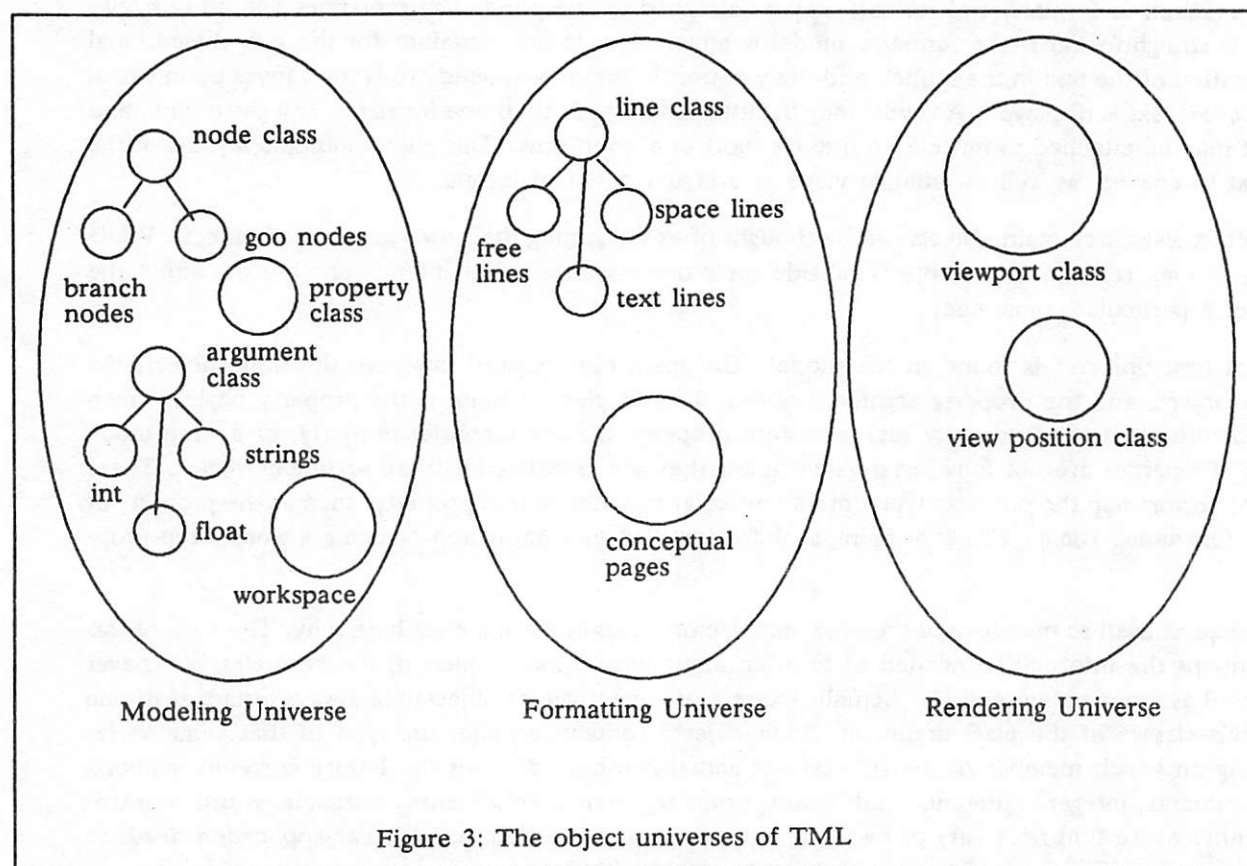
The objects used as members of the argument vectors actually form a class hierarchy. The base of this class contains the information needed to form an argument vector. Objects of the base class are never instantiated as independent entities. Actually existent argument vector objects are always instances of one of the sub-classes of the class argument. Such objects contain a value; the type of that value varies depending on which member of the sub-class is actually instanced. Thus the library currently supports string arguments, integer arguments, and floating point arguments. Other sorts of arguments will be introduced either as we find necessary or by the application developer when a particular application needs to hand values of a different type to a property entry or exit function.

The objects that make up nodes in the graph are also taken from a class hierarchy. The base members of this class contain the necessary pointers to locate them within the graph and a pointer off to a vector of properties. There are currently two subclasses of this class. The first is the branch node, which contains a pointer to its first child and is used for all non-terminal nodes in the graph. The second is the goo node, which contains a pointer off into the character heap and a count of the number of characters that are dominated by the node, these are used as the terminal nodes of the graph. We plan to expand this hierarchy far beyond its current state, adding nodes that contain graphic information of various sorts, nodes that refer to separate models, and perhaps nodes whose physical representation will be not graphi-

cal but audial. It is also this hierarchy that we expect to be expanded most often by application developers who can use it to introduce new sorts of text elements into their applications.

Within the formatting universe there are two main classes of object. The simplest of these is the conceptual page, which is basically an abstract representation of a two-dimensional object upon which the text is to be laid out. The contents of the conceptual page are conceptual lines. These are members of a class hierarchy; current members of the class include:

- Free lines, which indicate space on the page that may be replaced or partially replaced by other types of lines
- Space lines, which indicate space that has no contents, such as margins or gutters, that may not be replaced by other types of lines
- Text lines, which indicate space that is filled with characters in a particular font



As with the node objects, we expect the class of conceptual lines to grow considerably. This is no coincidence as the subclasses of conceptual lines are closely tied to the subclasses of nodes. A conceptual line is the product of a node formatting itself (or part of itself). This formatting is accomplished with a node class virtual function that creates a conceptual line. If the node being formatted contains text, the result of the formatting operation will be a text line. If we introduce another sort of node, such as one containing graphics, the result of formatting that node will be a different sort of conceptual line. The different types of lines are necessary because of the different ways the contents of the nodes will have to be rendered in the viewport; the actual drawing of a conceptual line is accomplished by a virtual function

of the class of conceptual lines. This drawing function can be tailored to the contents of the different sorts of nodes.

The simplest universe is that of the rendering component. Other than the viewport objects, this universe contains only the single class of entity that we call a view position. A view position is essentially a rectangular area of the viewport that contains a conceptual line. It's purpose is to allow translation from the coordinate space of the viewport to the coordinate space of the conceptual page.

As indicated above, the use of virtual functions in the node and conceptual line classes has allowed us to introduce a level of user extensibility that was lacking in the earlier, Pascal-based version of the toolkit. This extensibility allows us to add new sorts of node and line objects to the toolkit in the future, and also allows users of the toolkit to customize it for their particular application. The extensibility has also extended the very notion of what the toolkit can do. Originally, TML was designed to be a toolkit for text applications. We can now envision the addition of entities that encompass graphics and audio data, turning the toolkit into a hypermedia tool rather than merely a hypertext tool.

As mentioned above, the class structure of TML makes heavy use of the virtual function mechanism in C++. We were also able to exploit the function overloading capabilities in the language to provide a consistent syntax and semantics for operations across class boundaries.

One example of this can be seen by looking at the operation of attaching an object to another object. We saw above how models were attached to formats and formats attached to viewports. However, this operation is used in almost every class of object:

- Nodes, properties, and arguments are attached to models
- Arguments are attached to properties, which are in turn attached to nodes
- Properties are attached to other properties to form property vectors
- Arguments are attached to other arguments to form argument vectors
- Conceptual pages and conceptual lines are attached to formats
- Conceptual lines are attached to other conceptual lines
- Conceptual lines are attached to conceptual pages
- Conceptual lines are attached to view positions
- View positions are attached to viewports

In the Pascal version of TML, all of these operations were accomplished through distinct procedures. Because of this, an underlying similarity was masked. By using the overloading feature of C++, all of these operations are accomplished using what appears to be the same operation, the actual implementation of which is determined by what is being attached to what.

Our implementation also makes use of operator overloading to allow a natural ordering to be established on the contents of the graph that makes up a model. We introduced a model position object, which consists of a pointer to a node in the model and a byte offset into the data (if any) directly dominated by that node. Thus any character in a model is uniquely identifiable by a particular model position (although, in models that make use of the hypermedia facilities that we provide, a particular character may be identified by more than one model position). By overloading the boolean operations of greater-than, less-than, greater-than-or-equal-to, and less-than-or-equal-to, we allow the comparison of two model positions to

be based on where they would appear in the formatted document rather than where the nodes have been allocated in the object workspace of the model.

In short, the marriage of the design of TML and C++ as an implementation language has been a reasonably happy one. The facilities of the language have allowed the expansion of the library beyond the scope of its original design, and the features of the language have enabled the interface to the toolkit to present a unified view of the various universes of objects that make up the entities of text under the model we have adopted.

However, as with any marriage, not all is perfect. The problems we have run into range from mild irritations to significant difficulties. It is to these that we now turn.

Many of the minor irritations stem from the implementation of the language as a preprocessor. Because of this, the code that is generated is at least one step removed from the control of the programmer. While it is possible to debug the code and see the source as it was written, it is not possible to examine the contents of variables or set breakpoints using variable names or function names as they appear in the code. While it is in general not difficult to figure out the algorithm used by the preprocessor for generating the names, it can sometimes be a challenge when attempting to set a breakpoint on a function that is both virtual and overloaded. In much the same vein, the names generated by the preprocessor often exceed the 32-character limit found on many compilers. This generates a large number of warning messages in the best case; in the worst case it has required that we use overly terse member function names to insure uniqueness within the first 32 characters.

At the level somewhere between the irritating and the significant is the C++ construction *this*. The semantics of *this* is not particularly clear, especially the semantics within constructors. Furthermore, the implicit use of *this* within member functions can often lead to code that is difficult to decipher by any but the author. It is unclear, in such functions, whether a given name refers to a field of the object pointed to by *this* or refer to some global defined elsewhere. We have avoided this problem by simply not using global variables.

The most significant problem we have encountered, however, has to do with the lack of support in the language for saving objects to disk and restoring them later. The purpose of the model's object workspace is to locate all of the information concerning a particular text object in a place which will allow us to save the full text object and then restore it from disk without having to reconstruct the object from some linear data stream. The problem we have encountered is restoring the values of the virtual functions for the class that make use of such functions. Since the virtual function pointers point directly to code space, such pointers are not valid from one running of a program to another. In our case, the problem is compounded by the possibility that a text object may well be created and stored using one application and then read by some totally different application.

This problem is difficult but not insurmountable when confined to the basic objects in TML. It becomes considerably thornier, however, when considered in light of the prospect of applications adding new object types to a class, saving the object created to disk, and then attempting to restore that object using a different application (that may or may not know about the new type of object). It may well be that there is no general solution to this sort of problem. However, some thought should be given to improving the process of saves and restores.

## References

1. Lampson, B.W. **Bravo manual**. Alto User's Handbook, Xerox Corporation, Palo Alto, Calif., 1978.
2. Meyrowitz, N. and van Dam, A. **Interactive Editing Systems, Part 1 and 2**. *ACM Computing Surveys*, 14, 3 (1982), 321-417.
3. Nelson, T.H. **Getting it out of our system**, in *Information Retrieval: A Critical Review*, G. Schecter (ed.), Thompson Book Co., Washington, D.C., 1967, 191-210.
4. Waldo, J. **Modeling text as a hierarchical object**, in *1986 Summer Usenix Conference Proceedings*.

# The Design and Implementation of InterViews

Mark A. Linton and Paul R. Calder  
Stanford University

## Abstract

We have implemented an object-oriented user interface package, called InterViews, that provides **box** and **glue** classes that are used to compose a user interface from a set of interactive objects. The base class for interactive objects, called an **interactor**, and base class for composite objects, called a **scene**, define the protocol for combining several interactive behaviors. The InterViews library also provides common objects such as text, structured graphics objects, scroll bars, menus, and buttons.

InterViews is written in C++ and runs on top of the X window system. We are currently using InterViews to build an experimental programming environment.

## 1 Introduction

User interfaces are difficult to implement because of diverse user needs and preferences. Tools for programming a user interface are often too restrictive to build a variety of interfaces or too low-level to offer enough help to the programmer.

InterViews (*Interactive Views*) is a library of C++[5] classes that provides a set of powerful and flexible tools for constructing user interfaces. The design of InterViews was driven by two desires: (1) to provide a framework for implementing user interfaces without constraining the styles we could build, and (2) to make it convenient to build a user interface from smaller components.

Like Smalltalk MVC[1] and MacApp[4], the InterViews approach is to separate interactive behavior from abstract behavior. An interactive object, called a **view**, defines the user interface to an abstract object, called the **subject**. The separation of subject and view permits the use of different views of the same subject to customize interactive style. The user can dynamically customize this behavior using a **metaview**, a view of another view's internal state. For example, a text view may interpret keystrokes as commands using an internal mapping.

A text metaview might display this mapping and let a user modify it interactively.

The base class, **interactor**, defines the behavior of all interactive objects. All views are interactors, but not all interactors need be views; for example, it is more of a nuisance than a help to have a subject associated with a simple pop-up menu.

To build user interfaces from reusable components requires the ability to define an interactive object that is independent of its surrounding context. In particular, interactors must be flexible enough to handle variations in window size. The **scene** class defines the basic operations for managing a group of component interactors. One scene subclass, **box**, implements side-by-side composition of interactors using a simplified version of T<sub>E</sub>X[2] boxes and glue. Boxes arrange interactors to fit within a given area; glue specifies variable-sized space between interactors that is stretched or shrunk for a box of a given size. This model enables interactors to be composed within a box without specifying detailed layout information.

We have implemented InterViews on top of the X window system[3]. A small set of primitive classes completely encapsulate graphics and window operations. The remaining library classes and applications do not contain any X calls; they call operations defined on the primitives. This approach makes it possible to port InterViews quickly to other window systems.

## 2 Class Organization

Several factors influenced the structure of the InterViews class hierarchy. Our overall goal was simplicity: to make the classes easy to understand, straightforward to implement, and convenient to extend.

In order from most important to least important, the factors were:

### Shallow Nesting

Classes are a good partitioning mechanism, but there are drawbacks associated with a large number of classes. Our experience with an earlier library was that a large class hi-

---

Research supported by the SUNDEC project through a gift from Digital Equipment Corporation.

erarchy overwhelms programmers, especially when there are many levels of subclasses. The earlier library had many small classes nested 12 deep. Users of the library had trouble grasping all the different classes and all the inherited behavior. InterViews currently has one class that is nested 5 deep; most classes are at level 2 or 3. Throughout the evolution of InterViews we have generally attempted to add an operation to some existing class rather than to add a new class.

#### Many-to-Many Relationships

One reason for introducing new classes is to enable some state to be shared among several objects. For example, several interactors should be able to use the same graphics state; therefore, graphics state is a separate class. Similarly, several graphics states should be able to refer to the same font, so font is a separate class.

#### Common Usage

It is often preferable to design for a specific common case than for the general case. For example, InterViews does not define a unifying class for all kinds of menus. Instead, it provides one particular style, namely pop-up menus. The advantage is that a user need not understand a complicated menu model to use pop-up menus. The disadvantage is that there is no support for other kinds of menus, though they are straightforward to implement.

Figure 1 shows a subset of the InterViews classes. The top set of classes are all subclasses of **resource** because they are shared objects. Resources contain a reference count that can be manually incremented. When a resource is destroyed, the reference count is decremented but the resource is not deallocated unless the count is zero.

## 2.1 Interactors

All user interface objects are derived from the interactor class. Every interactor has an associated **shape** that it uses to define the desired display area characteristics, including natural size, shrinkability, and stretchability. An interactor's parent scene uses the shape when it allocates display space for the interactor. The actual display area is assigned to the **canvas** associated with the interactor. Because the

current implementation runs on top of X, canvases are always rectangular and may overlap.

When an interactor's canvas is set or changed in size, the interactor's **Resize** operation is called. If the interactor is a scene, it in turn assigns the canvases of its component interactors. The one exception is the root scene for a workstation display, called a **world**. A world's canvas is set to the entire screen when a program starts up. When an interactor is inserted into a world, its canvas is allocated immediately.

After a scene calls an interactor's **Resize** operation, it calls the interactor's **Draw** operation. The interactor should draw all of itself, including any components. When part of a canvas should be redrawn, perhaps because it had been obscured but is now visible, the interactor's **Redraw** operation is called with the coordinates of the affected area. In this case, the interactor should not redraw affected subcomponents; their **Redraw** operations will be called independently.

An interactor can perform output to its canvas using a **painter**. A painter provides drawing operations and manages graphics state such as foreground and background colors, font, and fill pattern. Each drawing operation is passed the target canvas. Canvas coordinates address pixels by default; however, coordinate values can be expressed in inches or centimeters by multiplying by the predefined global values "inch" or "cm". Also, painters can perform arbitrary coordinate transformations composed of translations, rotations, and scalings.

Associated with every interactor is a **cursor** that describes the visual representation of the pointing device (mouse) on the screen. When the mouse is over a visible part of the interactor's canvas, it is displayed as defined by the interactor's cursor.

An interactor can receive input events one of two ways:

1. It can read the next event on the (global) input queue.
2. An event can be passed from another interactor using the **Handle** procedure.

A **sensor** defines interest in certain kinds of events. Interactors interested in input events have a sensor that defines their current input interest. Each event is targetted to a particular interactor. The target is guaranteed to be interested in an event, but the reader may choose to process or ignore the event itself.

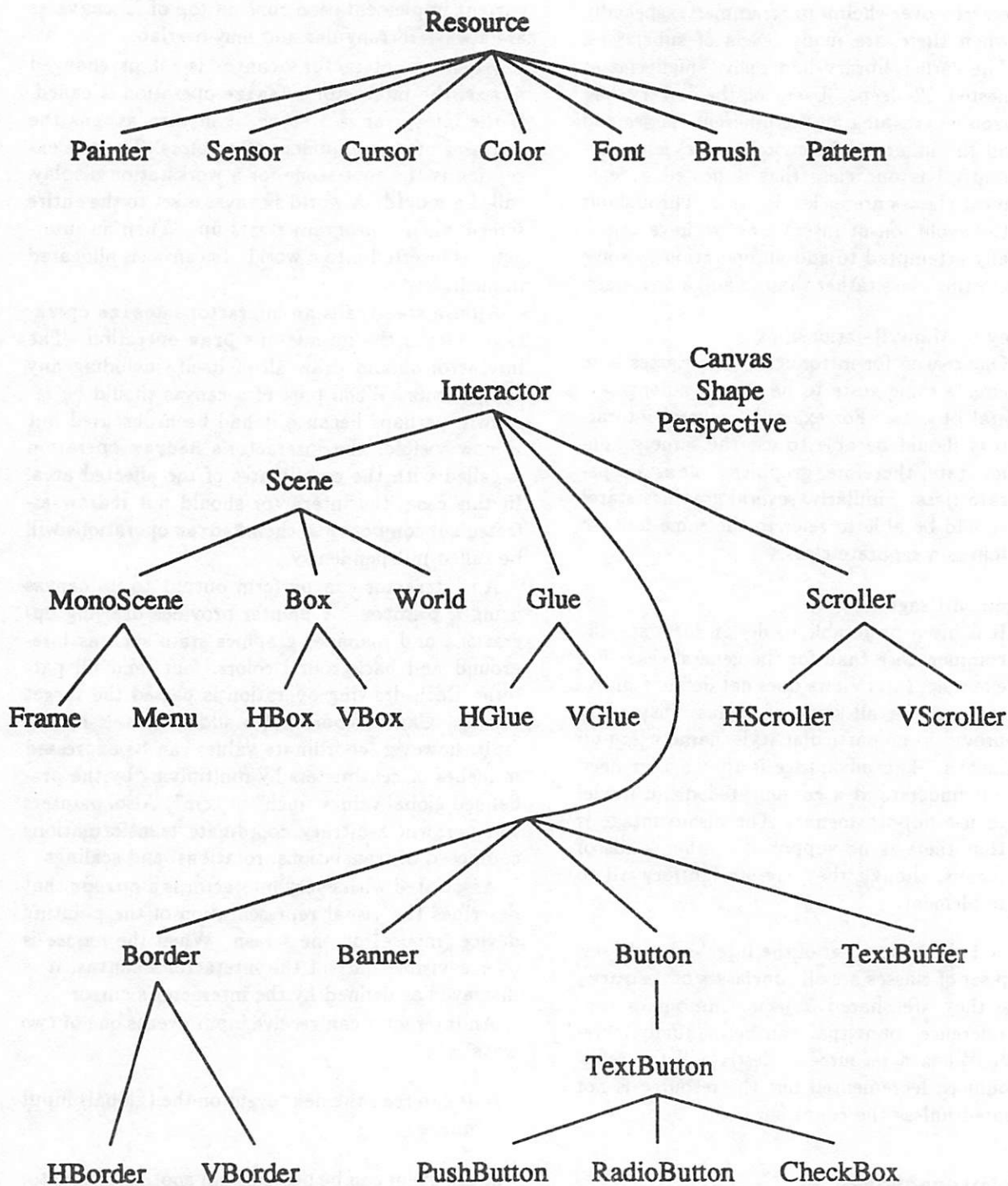


Figure 1: InterViews class hierarchy

A purely event-driven organization, such as in MacApp, can be produced by using the following C++ loop:

```
for (;;) {
    Read(e);
    e.target->Handle(e);
}
```

A more traditional control flow, which is not possible in pure event-driven systems, can be produced by reading events as part of interactor operations. For example, it may be desirable when a button is pressed in a pop-up menu to ignore events for targets other than menu items. In this case, having the menu interactor read events directly results in more straightforward code than in an event-driven implementation.

Because most interactors perform some output and are interested in some input events, every interactor has a sensor called "input" and a painter called "output". The initial values for input and output are defined when the interactor is created. Thus, the event interest and graphics state for most interactors is defined dynamically. Interactors can, of course, define additional sensors and painters.

Scenes often pass input and output to their component interactors, effectively sharing the state among several interactors. Because the state may be shared, it is inconvenient to make one particular interactor responsible for destroying the sensor and painter. The sensor and painter classes are therefore subclasses of resource. The interactor constructor explicitly increments the reference counts of input and output and the destructor decrements them. The object will only be deallocated when its count reaches zero.

Because we sometimes want to destroy an interactor without knowing which subclass it belongs to, we implement the concept of a "virtual destructor". Interactor subclasses do not define their own destructor; instead they define a `Delete` operation that is called by the base class destructor.

## 2.2 Scenes

All interactors that contain one or more component interactors are derived from the scene class. A common case is an interactor that is implemented in terms of another interactor. For example, a menu is implemented by a box containing the menu items. A menu does not share the behavior of a box in the sense of a subclass; it simply uses the box to com-

pose the items. This distinction is important, and it helps simplify the class hierarchy.

The scene subclass `monoscene` contains a single component interactor. A monoscene normally gives all of its display area to the interactor. One subclass of monoscene, `frame`, allocates all of its display space except for an outline around the interactor.

The most common scenes that contain several interactors are boxes. Two subclasses of box, `hbox` and `vbox`, arrange component interactors side-by-side horizontally and vertically, respectively. The box `Resize` operation shrinks or stretches each interactor according to the specification provided in the interactor's shape. A box tries to allocate each component interactor its natural size, then distributes any excess or shortfall in the available space according to the proportion of the total stretchability or shrinkability contributed by each component.

`Glue` is an interactor subclass that is used as a component in boxes. It has two subclasses, `hglue` and `vglue`, that are used to define horizontal and vertical space, respectively. A glue interactor therefore only shrinks or stretches along one axis, the same axis as that along which the box arranges components.

Thus, using boxes and glue we can define a presentation more flexibly. The layout can take on a variety of shapes and sizes without modification to the component interactors. More importantly, the interactors can appear in different layouts without being changed. By defining the natural sizes of components in a box, we can specify a default size for the box. By defining the relative stretchability and shrinkability of components, we can control the way in which they will be composed into the available space to maintain a pleasing or functional layout.

## 2.3 The World

Every program using InterViews must create a world object. This object represents the root scene of the display. The constructor opens a connection to the window server. In the X environment, it also specifies a display name. If the name is nil, then the value of the `DISPLAY` environment variable is used. The world object also provides an interface to the X defaults file and the standard X geometry specification syntax.

## 2.4 Scrollers

**Scroller** is an interactor subclass that displays a scroll bar. The **hscroller** and **vscroller** subclasses support horizontal and vertical scrolling, respectively. A scroller is a view of the **perspective** associated with another interactor. A perspective defines a range of coordinates and a subrange for the portion of the total range that is currently visible. For example, the vertical range for a text editor might be the total number of lines in a file; the subrange would be the number of lines actually displayed in the editor's canvas.

A scroller displays a scroll box whose length reflects the fraction of the total perspective that is currently visible. The user can modify the perspective interactively through the scroller using the mouse. The perspective can also be changed by the interactor itself, for example, if the text editor adds a line to the file. When an interactor changes its perspective, the perspective in turn notifies its views that it has changed.

Other kinds of views of perspectives are possible. A panner, for example, supports movement in both dimensions from a single interface. Zooming can be supported by changing the size of the perspective's subrange. More than one view of the same perspective can be created. For instance a perspective could be modified by zoom and scroll buttons in addition to the scroll bar.

## 2.5 Buttons

A **button** is an interactor subclass that is a view of a **button state**. The user can "press" a button to set the associated button state to a particular value. Several buttons can be visible for the same button state, making it possible to use buttons to select from a discrete set of values (each button represents a different value). Like any subject, when a button state changes value it notifies all the buttons attached to it.

Three common kinds of buttons are provided, all derived from a **text button** class that defines behavior for labelled buttons. A **push button** has a round-cornered rectangle surrounding its label. The rectangle and label are drawn in reverse colors (background and foreground swapped) when the button is pressed. A **radio button** has a circle to the left of its label: the circle is filled when the button is pressed. A **check box** has a rectangle to the left of its label: the rectangle has an 'X' in it when pressed.

In addition to being attached to a button state, buttons can be attached to other buttons. If button *A* is attached to button *B*, then *A* is disabled while *B* is not pressed. A disabled button ignores input and draws itself with a gray pattern to show it is disabled.

## 2.6 Other Classes

InterViews includes a number of other classes that support user interfaces. A **text buffer** manages a two-dimensional array of characters. **Page** provides hierarchically structured text objects together with composition objects for a variety of layout styles.

A **banner** displays left-justified, centered, and right-justified headings. A **border** fills an area of a certain thickness; a **vborder** can be stretched vertically and the thickness determines its width; a **hborder** can be stretched horizontally and the thickness determines its height.

A **menu** is a box of **menu items**. When its **Popup** operation is called it inserts itself into the world, waits for the user to release a button, and returns the menu item that was chosen.

**Picture** is a base class for defining structured graphics objects. Subclasses of picture include basic graphics objects (such as **line**, **circle**, and **rectangle**) and **group** for representing a collection of pictures. **Picture** is a subclass of a **persistent** class so that picture data structures can be automatically stored on disk.

**Rubberband** is a base class for graphics objects that track user input. For example, a rubber rectangle can be used to drag out a new rectangle interactively. Another subclass, sliding rectangle, can be used to move around an existing rectangle. These classes completely isolate programmers from the device-dependent use of exclusive-or drawing or the use of an overlay plane.

**WorldView** is a base class for defining window managers. It provides operations that should only be used by a window manager, such as controlling input focus between several applications.

Workstation parameters are accessed using a pre-defined global object named **workstation**. The object is created when the associated world is created and can return information such as the physical dimensions of the display.

## 3 Example Usage

**Squares** is a demonstration program that uses

many of the *InterViews* classes. The program contains a simple subject that manages a list of squares of different sizes and positions. The user interface is constructed from a view of the squares list, a frame around the view, and a dialog box for simple customization.

The frame surrounds a vertical box containing a banner and two horizontal boxes, all separated by horizontal borders. The upper horizontal box contains the squares view, a vertical border and a vertical scroller. The lower horizontal box contains a horizontal scroller, a vertical border and a piece of glue. Figure 3 shows what a squares frame looks like; Figure 2 shows the C++ code that constructs the frame.

Using a pop-up menu, the user can create another view of the squares list, add a square to the list, open a dialog box to customize the squares frame, or exit the program. Figure 4 shows the result of creating a second view and adding a square. The squares list notifies its views when the square is added, so that the new square is visible in both.

Figure 5 shows the dialog box used to customize the frame around a view. The dialog box contains check boxes for specifying the presence of scrollers, buttons for specifying attributes of the scrollers, and a confirmation button to indicate that customization is complete. The components of the dialog box are separated by glue objects. The glue above the "Horizontal Scroller" check box has high shrinkability, while the glue between the check box and the radio buttons has low shrinkability. If the vbox containing these objects is not given enough screen space to fit the natural sizes of their views, it will shrink the glue above the check box more than the glue below it. Figure 6 shows a resized dialog box.

## 4 Implementation

It took about three man-months to implement *InterViews* on top of X. In this section, we discuss some of the problems interfacing to X, the details of implementing boxes, and some reflections on using C++.

### 4.1 Interfacing to X

*InterViews* primitive class operations make direct X library calls to implement their semantics. The two main problems interfacing to X were managing

windows and translating X input events into *InterViews* events.

#### 4.1.1 Window Management

Each canvas is represented as an X window. The world's canvas is the root window for a display. The scene class contains operations to perform the creation, mapping, and configuration of windows. The two operations available to subclasses are *Place* and *UserPlace*. *Place* puts an interactor at a specific position in a scene, which is implemented by creating a subwindow of the scene's window and associated the subwindow with the interactor's canvas. *UserPlace* creates a window and lets the user interactively position it.

#### 4.1.2 Input Events

The X model of input events is slightly different from the *InterViews* model. An important similarity is that associated with each X input event is a destination window. The interactor *Read* operation maps the window to an interactor through a global hash table maintained by scenes. The event is then checked against the interactor's current sensor to see if the interactor is interested in the event. Normally, we can tell X to ignore events that are not of interest; however, X cannot always distinguish events at the level we wish. For example, X cannot send events for the left mouse button and ignore events for the middle and right buttons.

X is very different from *InterViews* in the sizing and redrawing of windows. X represents the need to redraw part of a window as an input event; *InterViews* represents it as an out-of-band procedure call. When the *Read* operation sees a redraw event, it calls *Redraw* on the destination window and proceeds to read the next input event. Thus, a single *InterViews* read can result in reading multiple X events.

The normal approach to processing X redraw events has an inherent performance problem. If a large number of windows need to be redrawn (in a hierarchy, for example), a large number of redraw events will be received. The result is a loop consisting of reading an event and redrawing part of a window in response. Because reading an event causes all pending output to be flushed, this loop defeats the X library buffering strategy and effectively increases the cost of every drawing operation. We solve this problem by storing redraw events on a queue instead of performing them directly. When

```

frame = new VBox(
    banner,
    new HBorder(output),
    new HBox(
        view,
        new VBorder(output),
        new VScroller(view, vwidth, nil, output)
    ),
    new HBorder(output),
    new HBox(
        new HScroller(view, hwidth, nil, output),
        new VBorder(output),
        new HGlue(output, vwidth, 0)
    )
);

```

Figure 2: Code to construct squares frame

an event read will block or the event is not a redraw, all of the redraw events on the queue are performed.

## 4.2 Boxes

An interesting aspect of implementing boxes is the computation of the shape of a box. A box must compute its own shape as a function of the shapes of the interactors inside it. Along the major axis (horizontal for an hbox, vertical for a vbox), the natural sizes, stretchabilities and shrinkabilities can simply be added.

Computing the parameters for the minor axis is more complicated. The natural size is the maximum of the component sizes. The minimum size is the maximum of the component minimum sizes, computed by subtracting their shrinkability from their natural size. The shrinkability of the box is therefore its natural size minus its minimum size.

The maximum size of the box is the minimum of the component maximum sizes, computed by adding their stretchability to their natural size. The stretchability of the box is therefore its maximum size minus its natural size.

## 4.3 C++ Experiences

Using C++ as the implementation language for InterViews has had several advantages and disadvantages. Inheritance and virtual functions simplify the structure of code and data that would otherwise need to use function variables and additional levels of indirection. C++ is also very portable, enabling

us to bring up InterViews on Sun workstations in a matter of hours.

The major disadvantages we have found in using C++ are that it does not support multiple inheritance (yet) and the run-time system does not support garbage collection. Multiple inheritance is necessary for a clean class hierarchy and to facilitate shared behavior. For example, we have a subclass of frame, called **title frame**, that has a banner at the top. We cannot create a kind of banner that contains several interactors, because banner is not a subclass of scene. We cannot create a subclass of scene and use it in a title frame because it would not be type-compatible with banner. It would be incorrect for banner to be a subclass of scene, because the basic banner does not contain component interactors. With multiple inheritance, we could define a class that derived from both scene and banner.

Garbage collection would be very helpful because of object sharing. For a variety of reasons, it is beneficial to share objects in a user interface implementation. Sometimes the sharing is for the purposes of using the same output style; sometimes it simply eliminates the need to create new objects. When objects are shared, it becomes difficult to decide when the object should be destroyed. Our resource class solves the problem for a specific set of classes, but requires manual reference counting.

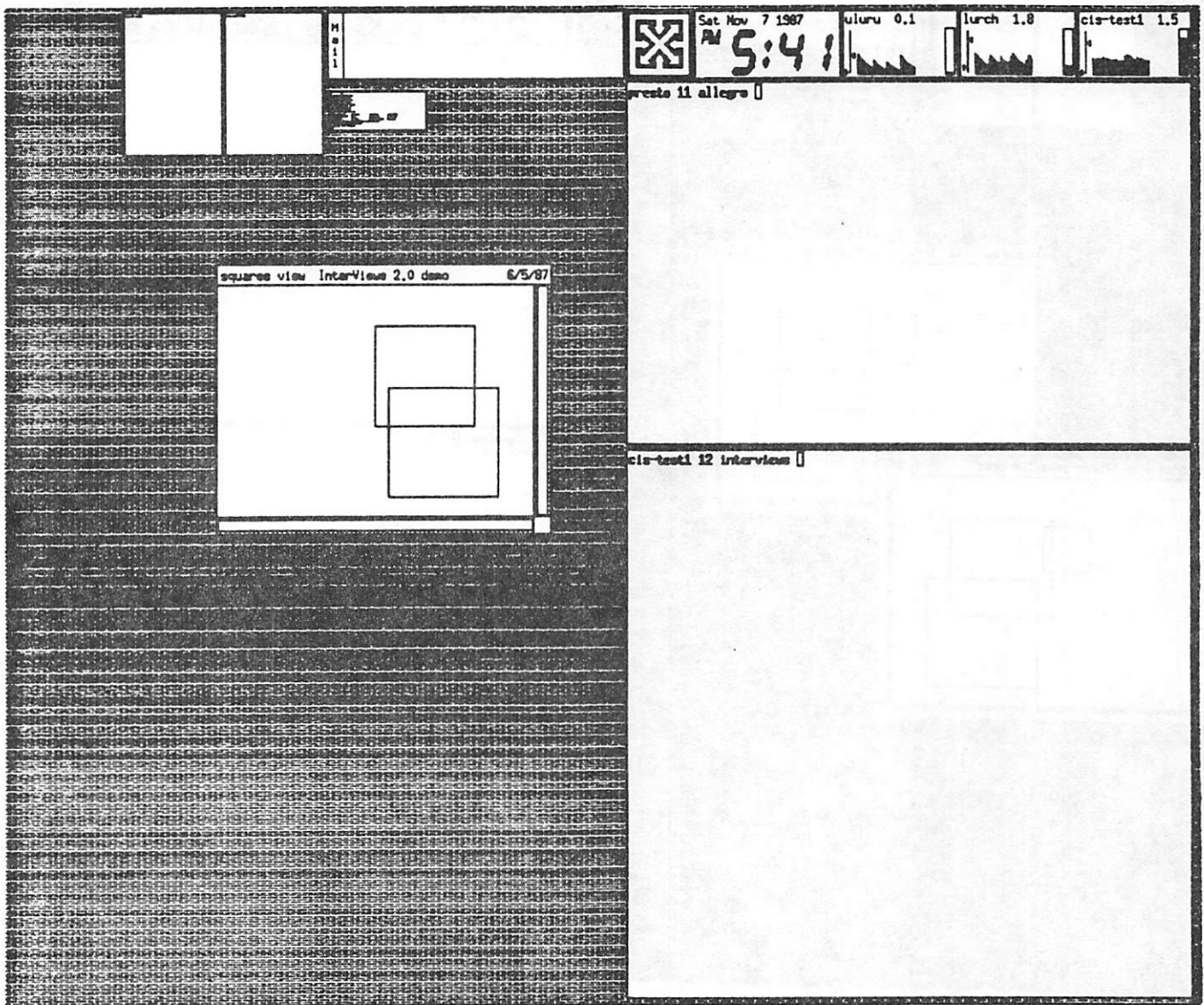


Figure 3: Screen dump containing squares view

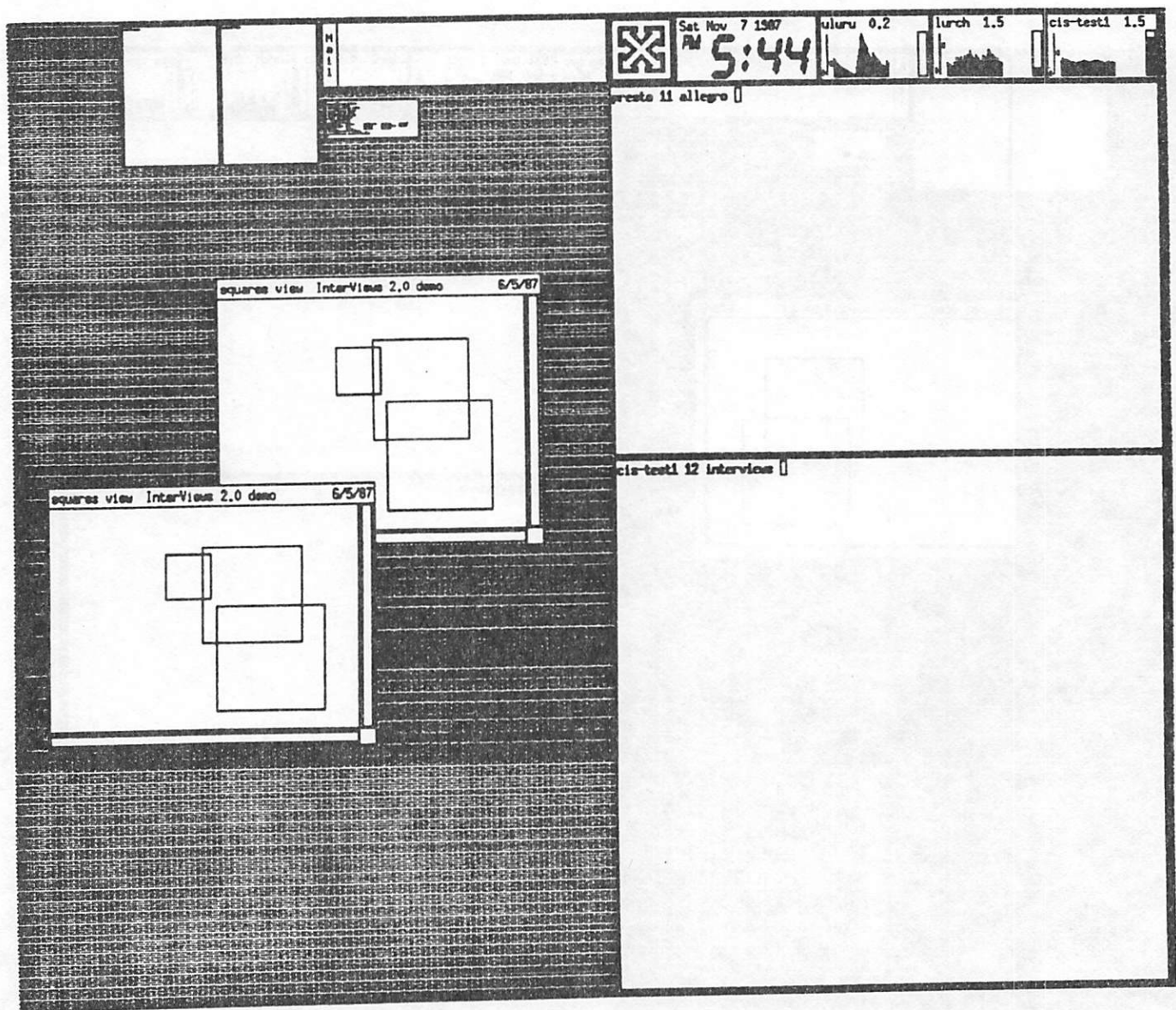


Figure 4: Second squares view

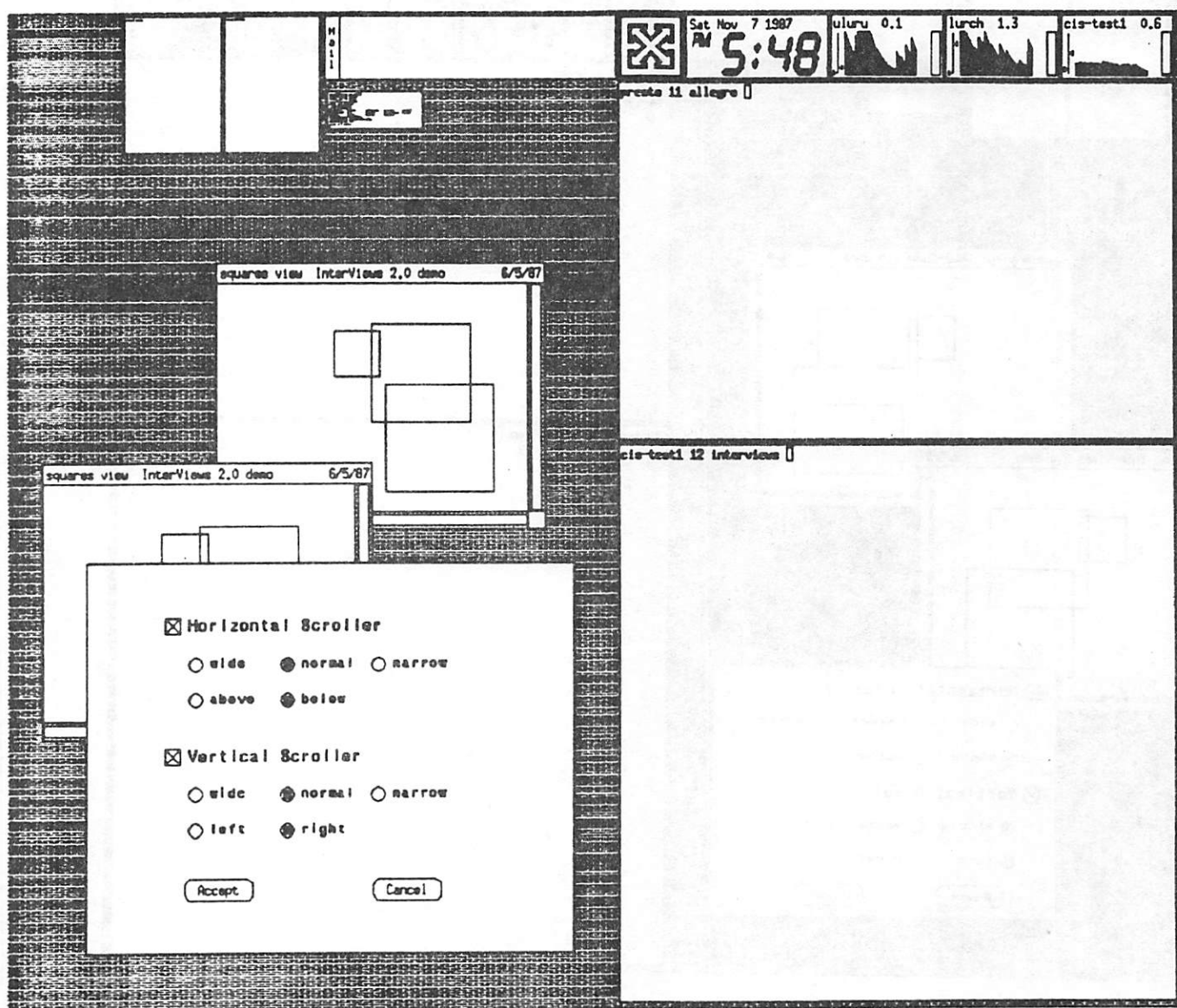


Figure 5: Squares metaview

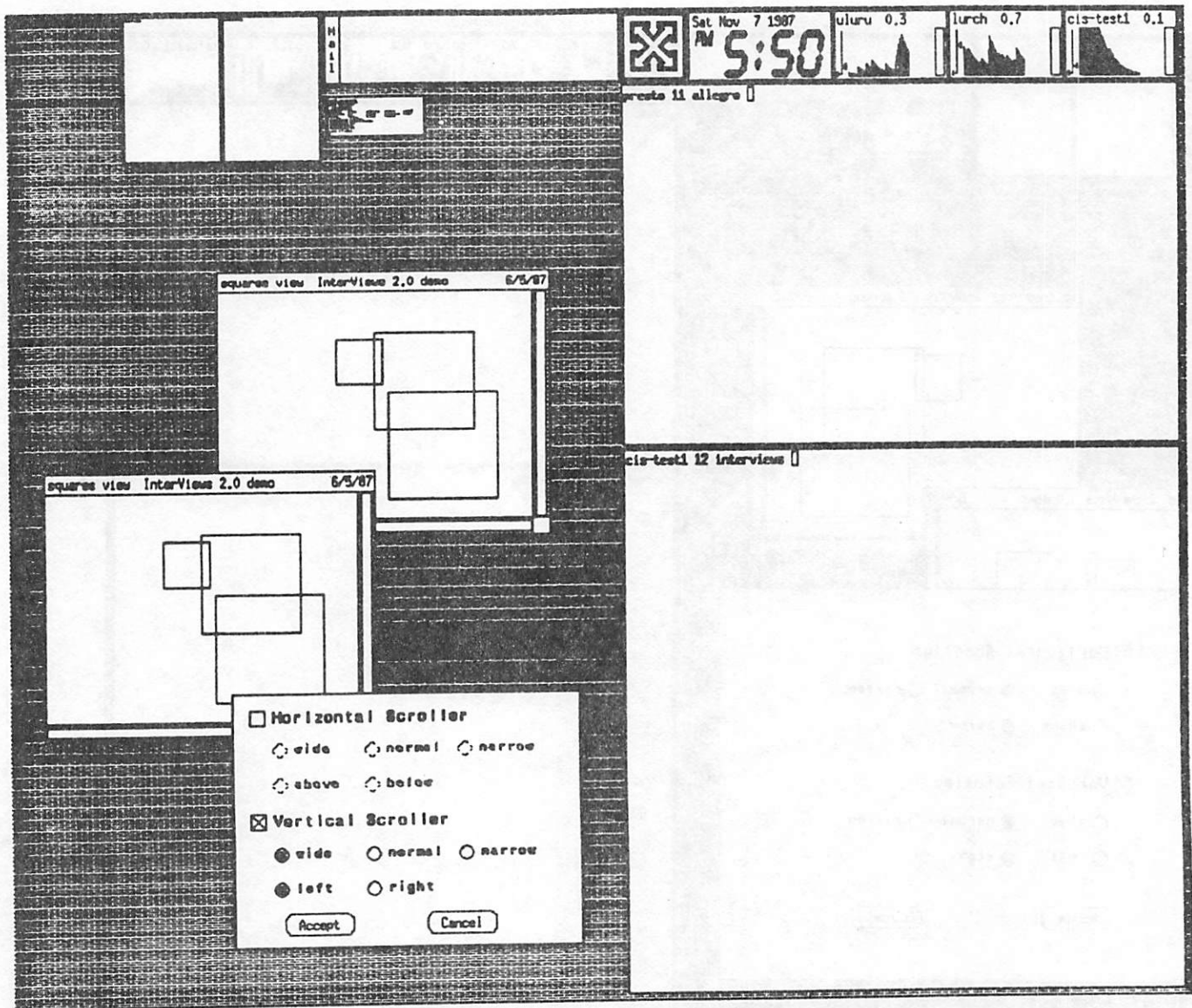


Figure 6: Resized metaview

## 5 Current Status

InterViews currently runs on MicroVAX and Sun workstations on top of either X10 or X11. The library is roughly 15,000 lines of C++ source code. We have also implemented several applications on top of the library, including a reminder dialog box, a scalable digital clock, a drawing editor, a load monitor, a window manager, and a display of incoming mail. We are currently working on a more general drawing system, a structure editor, and a visual debugger as part of a C++ programming environment.

## 6 Conclusions

InterViews provides a relatively simple organization of user interface classes that is easy to use and extend via subclassing. The use of boxes and glue make it possible to define reusable user interface components.

The InterViews library completely hides the underlying window system from application programs. Our original reason for having this layer was that our model was different from X's, but in retrospect this approach has the added benefit that we can port InterViews applications to a new window system simply by porting the primitive classes.

## Acknowledgments

Craig Dunwoody and John Vlissides participated in the design of InterViews. John implemented the drawing editor, the structured graphics classes, and coordinate transformations in the painter class. Paul Hegarty implemented the window manager. John Interrante helped write the reference manual.

## References

- [1] Goldberg, A., *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, Massachusetts, 1984.
- [2] Knuth, D., *The T<sub>E</sub>Xbook*. Addison-Wesley, Reading, Massachusetts, 1984.
- [3] R.W. Scheifler and J. Gettys. "The X Window System", *ACM Transactions on Graphics* Vol. 5, No. 2, April 1987, pp. 79-109.
- [4] Schmucker, K. J., *Object-Oriented Programming for the Macintosh*. Hayden. Hasbrouck Heights, New Jersey, 1986.
- [5] Stroustrup, B., *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 1986.

# The Design of the Allegro Programming Environment

Mark A. Linton, Russell W. Quong, and Paul R. Calder  
Stanford University

## Abstract

The Allegro programming environment uses a *distributed object-oriented* organization for software information and tools. A program is represented as a collection of objects, which are grouped into several *object spaces*. Operations can be performed transparently on objects located in a remote space. Program objects explicitly store dependencies on other objects to facilitate source browsing and incremental compilation.

We are implementing Allegro with the goal of using it ourselves for further Allegro development. In this paper, we present the Allegro architecture, show how to implement transparent object communication, describe the representation of program objects, and discuss an approach to fast turnaround in a compiled environment.

## 1 Introduction

Improving programmer productivity means decreasing the time it takes to produce a software system. Software development is often *evolutionary*; that is, programmers debug and enhance a system over time.

The goal of the Allegro programming environment is to reduce the time it takes to evolve code. We are interested in "production" rather than "exploratory" software development. In particular, we assume a compilation-based environment, large target systems, source code distributed across a network of machines, and the use of multiple languages.

Allegro manages program information using a *distributed object-oriented* architecture. By "object-oriented" we mean that all program information is represented in terms of objects that each have some associated state and belong to a class that defines the set of operations for accessing and manipulating the state. By "distributed" we mean that objects can transparently reference and perform operations on objects located on remote machines.

Research supported by the SUNDEC project through a gift from Digital Equipment Corporation.

An object in Allegro is similar to an object in a programming language such as Smalltalk[6] or C++[7]. Modules, variables, statements, expression nodes, relocatable addresses, and execution breakpoints are all examples of objects in Allegro. The differences between Allegro objects and Smalltalk or C++ objects are (1) Allegro objects can be distributed throughout a network, and (2) remote object communication is independent of any specific inheritance or typing mechanism.

Because Allegro objects are too small and plentiful to be implemented as distinct files or processes, we group a number of related objects into an *object space*, which is like an address space with objects taking the place of bytes or words. As with address spaces, the criteria for partitioning objects into spaces depends on specific needs for protection, naming, efficiency, and size limitations. Just as an address space represents data that is logically in memory but might be temporarily stored on disk, an object space represents objects that are logically in memory but might temporarily be swapped out until referenced. Each object is responsible for storing itself on disk when appropriate.

Representing program information as objects has two important benefits. First, an object-oriented representation allows decentralized management of software information. Common access methods can be implemented independently for different internal representations, so performance can be tuned for individual operations.

Second, an object-oriented representation simplifies the design and implementation of an incremental compilation system. An incremental system compiles and links a program in time proportional to the size of a change; a batch system runs in time proportional to the total size of a program. One of our goals for Allegro is to provide the fast turnaround normally associated with interpreted environments such as Smalltalk for a compilation-based environment such as Unix<sup>†</sup>[4]. Objects lend themselves naturally to incremental algorithms because an object's state can contain information necessary to perform an update efficiently. A common

<sup>†</sup>Unix is a registered trademark of AT&T Bell Laboratories.

technique is to maintain a list of the other objects that should be processed when an object is modified.

Measurements of program development on Unix indicate that most changes involve one or two modules and that modified modules usually do not change in size much if at all. Thus, incremental techniques can achieve fast turnaround after a change. We have implemented an incremental linker that is more than an order of magnitude faster than the standard Unix linker.

Our implementation of the Allegro object space model demonstrates that communication among objects in different spaces can be implemented efficiently. We purposefully have avoided three issues of a distributed object-oriented system: global garbage collection (across spaces), high reliability, and transaction management (especially involving multiple object spaces). While these problems are important, we have chosen to focus our efforts on the environment architecture, program representation, and support for incremental algorithms.

## 2 Allegro Organization

Objects in Allegro are organized into object spaces according to two criteria:

1. Objects with many interrelationships that as a whole represent a more complex object are usually stored in the same object space. For example, objects representing the source code for a program, including procedures, statements, expressions, and types are generally in the same space.
2. Object spaces are also organized hierarchically just as files are in a hierarchical file system. For example, "/usr/linton/xyz/src" might refer to the object space containing the source objects for the program "xyz".

Objects in one space can refer to objects in another space, so object spaces can be organized to share information the same way files are organized. For example, programs "xyz1" and "xyz2" could have references to objects in a subroutine library represented by a separate object space. References across spaces make the organization of objects a graph, not a pure hierarchy.

From the programmer's point of view, Allegro is thus similar to environments such as Unix. Programs are structured hierarchically, and source,

documentation, binary, and executable objects are separate leaves of the tree. However, much more information is readily available in Allegro objects than in Unix text files.

Figure 1 shows an example of the object spaces for a programming session. The **allegro** space implements the programming environment user interface and coordinates the other spaces. The **source** space is an interface to source objects for the program, which can either be edited using a structure editor (**insted**) or a text editor. The only way to access source code is to perform operations on source objects: files are not visible to other object spaces.

The **stabd** space contains symbol table objects, including type and run-time address information. The **inlink** space contains the global symbol and relocation information needed to create an executable. When a module is compiled, a new object file is created and both **stabd** and **inlink** are notified. Each space reads the new object file and updates its internal state. **Inlink** updates the program executable with the changed modules.

The **ptraced** space implements process and stack frame objects for debugging a sub-process. Because **ptraced** is an independent object space, it can be on a different machine from the other processes. The **ptraced** space also encapsulates machine and operating system-dependent debugging facilities. The **ttyhandler** space filters standard input and output from the running program to the user interface.

## 3 Implementation Issues

At the heart of Allegro is a simple, efficient mechanism for object communication. This mechanism is similar to a remote procedure call[1], but it also supports references to objects and batching of multiple operations into a single message. Above the communication layer are the objects that represent program information. Program source is represented in an object-oriented intermediate language called SLIC (Source Level Intermediate Code) that explicitly represents dependencies.

### 3.1 Object Communication

An operation on a "local" object (in the same object space as the caller) is essentially a procedure call. For communication to be transparent, an operation on a "remote" object (in a different space) must

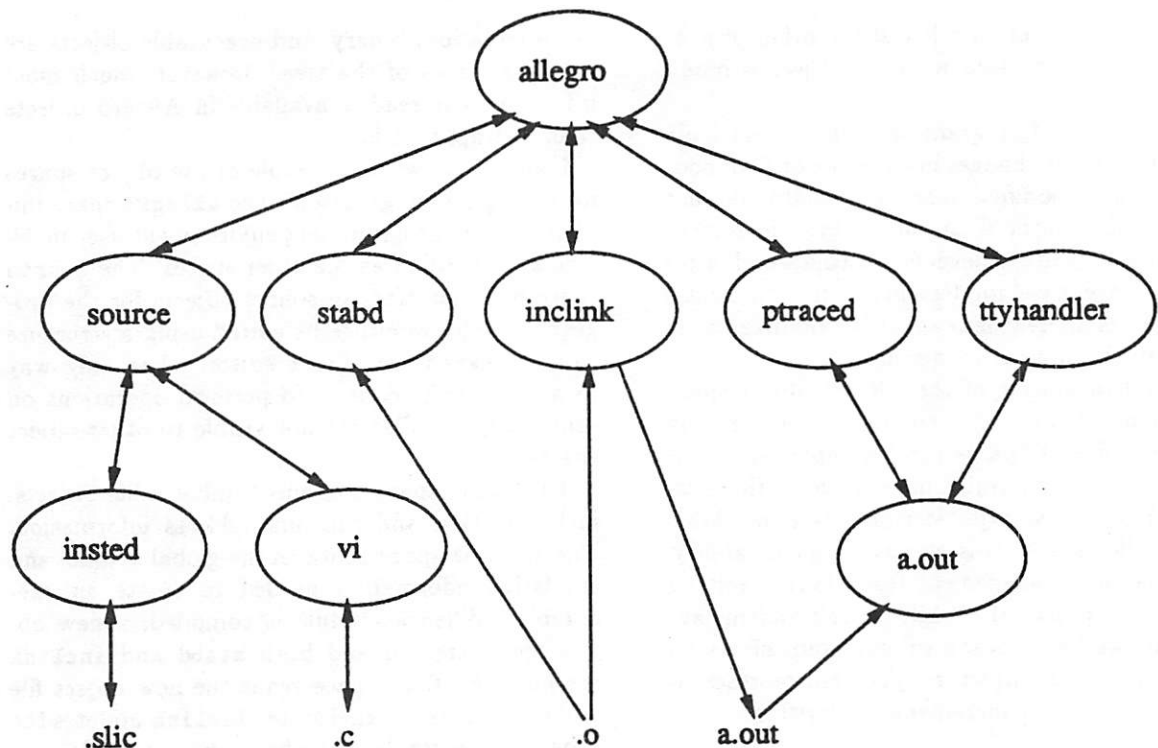


Figure 1: Allegro object spaces

also be a procedure call. To achieve this uniformity, we associate a special local object, called a **deputy object**, with the remote object. A deputy packs an operation into a message; a **stub** object unpacks the message.

Because it is expensive to establish a connection between two objects for every operation, each object space has a set of **chief deputy** objects that coordinate outgoing messages and a set of **messenger** objects that handle incoming messages. Each chief deputy sends messages from deputies over a connection to another object space: each messenger receives messages from an object space and forwards them to the appropriate stubs.

Figure 2 shows an example with two objects in distinct object spaces. Object *A* is located in object space *I* and object *B* is in space *J*.  $B_I$  is the deputy for *B* in space *I*;  $B_J$  is the stub object for *B*. When *A* performs an operation on  $B_I$ , the deputy creates a message containing *B*, the operation, and any parameters. The chief deputy for *I* sends the message to *J*, where the messenger associated with *J* takes the message and forwards it to  $B_J$ .  $B_J$  unpacks the message and performs the operation on *B*.

The chief deputy and messenger hide the operating system details of inter-process communication. They also reduce the cost of communication by transmitting multiple operations in a single packet when possible. If the semantics of an operation let the caller continue while the operation is being performed, the operation can be buffered by the chief deputy. We call such an operation a **command**. The chief deputy sends the buffer when it is full or when an operation is performed that must be executed immediately. When a buffer of commands is received, the messenger breaks up the buffer and delivers each operation to its appropriate stub.

### 3.1.1 Naming

A deputy must be able to establish a link to a remote object and refer to the remote object in messages in a way the messenger understands. The link is established by using a character string name, and a 32-bit tag is used to refer to a remote object.

The strategy for mapping names to objects is based on the concept of decentralized naming in the V System[2]. The idea is that instead of a central name-to-object mapping, each object space has

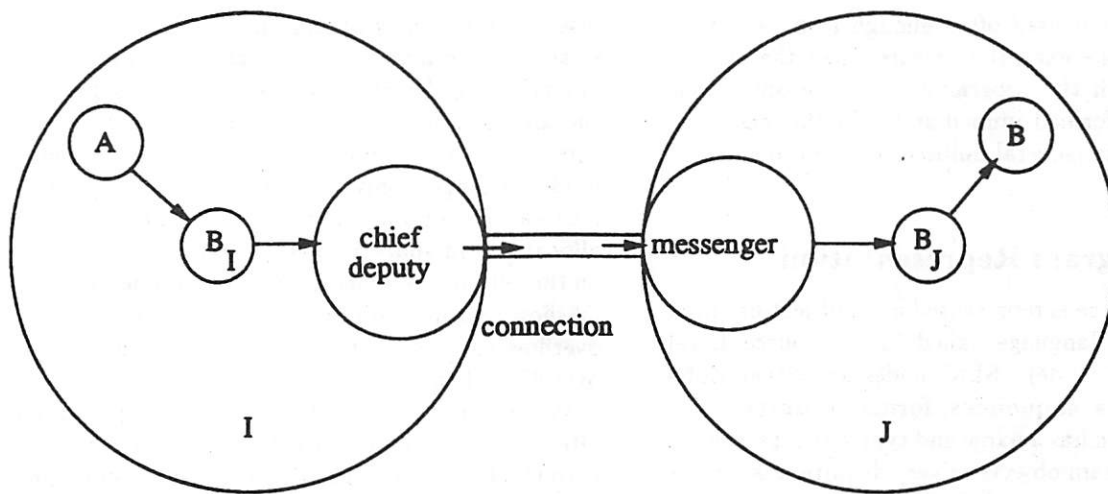


Figure 2: Communication across object spaces

its own catalog for mapping names of objects inside it. Object spaces themselves are leaves in a directory hierarchy. To find the object associated with a full pathname, a request is sent to all the object spaces. The one that responds is the one that contains the name. In practice, most queries can be avoided by caching which object space contains which name prefixes. Measurements on the V System indicate that for intra-machine and local network access, this approach is more efficient than centralized name resolution.

In our implementation, sending a request to all object spaces is prohibitively expensive because Unix does not have a multicast send primitive. Instead, a message is sent to an object space manager associated with the host machine. If a requested name refers to a space on the host, the space manager returns a local connection to it. Otherwise, the space manager queries the space managers on other machines and returns a remote connection to the named space if it can be found.

Once a connection to the object space is obtained, a special request is sent containing the name of the requested object and the 32-bit tag that will refer to the object in future messages. The messenger interprets this request itself, rather than forwarding it to a stub. If the messenger finds the name in the catalog for the object space, it creates an entry in an object table that associates the tag with the object's stub. The messenger uses this table to translate

tags to stubs in later messages.

The tag is defined by the deputy rather than by the messenger for two reasons:

1. If the tag is defined by the deputy, then the messenger need not reply when a deputy is created. This convention lets object creation be defined as a command and thus buffered.
2. If a reply containing a reference to the object is sent back to the deputy, the tag can be interpreted easily (it could be a memory address, for example).

If a remote object is created as the result of an operation on another remote object, it need not have a name. This case occurs when a new remote object is created as a result of an operation on other remote object. For example, it is common to access a remote prototype object by name and copy it to generate new instances.

### 3.1.2 Performance

The overhead for communication across object spaces is small (under 10%) even though the cost of sending a message is relatively expensive (roughly 3 ms on a MicroVAX-2 running Unix). The overhead is low because of the careful design of object interfaces. Operations that can be accessed remotely are commands (and are thus buffered), execute much longer than the cost of sending a mes-

sage, or are not used often enough to be a bottleneck. The one exception occurs when the system interacts with the programmer, where operations must be performed immediately. In this case, the absolute delay (several milliseconds) is too small to be noticed.

### 3.2 Program Representation

Program source is represented in an object-oriented intermediate language called SLIC (Source Level Intermediate Code). SLIC nodes are either **definitions**, **uses**, **sequences**, **forms**, or **units**.

A definition has a name and typically represents a class of program objects. Every definition is associated with a form that defines the components of its instances, how to display and manipulate instances, and how to generate a source file for a compiler. Each definition also contains a list of its uses.

A use represents a program object; it is associated with a definition. A sequence is an ordered list of uses. A unit is a logical group of program objects, corresponding to a module in the program.

SLIC objects can either reside in memory or on disk. In memory, an object is dynamically allocated and contains references to other memory-resident objects. On disk, an object is represented by a stream of text with symbolic references to other disk-resident objects. The SLIC textual representation is designed to be easily processed. The representation is much like a stack virtual machine, except that the result of a "computation" is to build the corresponding in-memory representation.

The **insted** object space uses a SLIC representation for programs. In addition to implementing SLIC objects, **insted** implements an interactive WYSIWYG editor for SLIC objects.

### 3.3 Fast Turnaround

We have profiled how programs are built in our environment by instrumenting the Unix **make** utility[3] and found that most changes are small. The details of this study are given in [5]. Typically, programmers edit one or two modules, recompile them, relink the program, execute the program, find errors, and repeat the whole process. Only a small part of the program actually changes from version to version of the executable, even for large programs with many modules. Thus, incremental techniques can significantly reduce turnaround time.

The **stabd** and **inlink** objects incrementally process program changes. The **stabd** objects rep-

resent symbol information on a per-module basis, so that a new module can replace an old one without affecting the other modules. Every symbol in the **inlink** space has a list of addresses to relocate when the symbol's address changes. When a module changes, only the affected addresses are relocated. To minimize updates to the executable, we allocate extra space for every module. Simulations on the actual compilation and link data described in [5] show that if we allocate 25% of the executable for overflow space we can update the module in place over 95% of the time.

We measured the CPU time to link a program after a small change using **inlink** and compared it to the Unix linker, **ld**. For a relatively small program (5,000 source lines), linking incrementally is about 10 times faster than **ld**. For a large program (100,000 source lines), it is 70 times faster.

This experience indicates that we can reduce to nearly zero the time from when modules have been compiled to when execution begins within a debugging environment. We are investigating several techniques for reducing compile time:

- A language-oriented editor can perform much of the work of a compiler as part of editing.
- Compiling modules in background after they are edited can use available cycles during think time.
- Parallel compilation of the set of changed modules can use available cycles on other processors.

## 4 Current Status

We have implemented a run-time library that supports remote object communication. This library includes the **deputy**, **chief deputy**, **messenger**, **object space**, and **object stub classes**. The implementation is written in C++ and uses the Berkeley Unix inter-process communication facilities to implement connections between object spaces.

The **inlink**, **stabd**, and **tythandler** object spaces have been implemented using the run-time library. We are currently working on the **ptraced**, **allegro**, and **insted** spaces. Our initial program representation is targetted at C++, so that we will be able to use Allegro ourselves. We are also implementing a parser for C++ so that we can create **insted** objects for existing code.

## 5 Conclusions

Object spaces unify the traditional concepts of commands and files. Unlike files, which are nominally on disk but cached in memory, objects are normally in memory but stored permanently on disk. Allegro therefore explicitly trades memory for faster access time without any caching mechanism. The added payoff of this organization is that it facilitates the implementation of incremental algorithms, which can significantly reduce turnaround time.

The disadvantage of this design is that each object space is more complex than a traditional software tool because of the support needed for transparent object communication, locking, and query processing. Our experience so far is that this complexity can be shared in the run-time library. Only when we are using Allegro for day-to-day development will we really know the practicality and effectiveness of this approach.

## References

- [1] A.D. Birrell, B.J. Nelson, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1984, pp. 39-59.
- [2] D.R. Cheriton and T.P. Mann, "A Decentralized Naming Facility". Technical report STAN-CS-86-1098. Computer Science Department, Stanford University. February 1986.
- [3] S.I. Feldman, "Make - A program for Maintaining Computer Programs". *Software Practice and Experience*, Vol. 9, No. 3, March 1979, pp. 255-265.
- [4] B. Kernighan and J. Mashey, "The Unix Programming Environment", *Computer*, Vol. 14, No. 4, April 1981.
- [5] M.A. Linton and R.W. Quong, "A Macroscopic Profile of Program Compilation and Linking", Technical Report CSL-TR-97-313. Stanford University. Stanford, California, February 1987.
- [6] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, Massachusetts. 1984.
- [7] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts. 1986.
- [8] W. Teitelman and L. Masinter, "The Interlisp Programming Environment". *Computer*, Vol. 14, No. 4, April 1981, pp. 25-33.

# A C++ Class Browser

Raghunath Raghavan, Niranjana Ramakrishnan & Sue Strater

*Mentor Graphics Corporation  
8500 SW Creekside Place  
Beaverton, OR 97005*

## 1. Introduction

Mentor Graphics has been one of the earliest users of C++ (outside AT&T). For some time now, the need for a mechanism to keep track of the classes produced has been quite apparent. This mechanism is a C++ class browser.

Lately, the number of users of C++ within the company has been increasing dramatically. Frequently, new developers have been duplicating existing functionality, either because they were unaware of its existence or because its existence was buried in some base class in some other file.

Object-oriented programming encourages a development style that re-uses and/or extends existing code. Thus, the user is constantly browsing through the directories where existing classes are declared. Up to now, the usual method of browsing existing classes has been to look in some standard directories or to ask the 'old-timers'. This involves groping through the file system, opening and closing files.

Even after finding the required class declaration, there remains the problem of finding the relevant information. Class declarations tend to be heavily documented/commented. The existence of large blocks of comments interferes with getting an overall view of the functionality of a class. To make matters worse, part of the functionality may lie in base classes that are declared in some other file. Also depending on the client's view, some information is not relevant, for example if a client only needs to see the public interface.

While there are standard file templates to help read a C++ class declaration, enforcing the standard and ensuring adherence to it is not an acceptable long-term solution to the problem.

A better approach is to build a C++ class browser that would present the existing class libraries in a meaningful way to class clients. There are two types of class clients:

- clients who want to use a class as-is, and need to know the complete public interface, and

- ❑ clients who want to derive from an existing class, and must be aware of the complete protected and public interfaces, especially virtual functions. (A source of errors is a derived class unwittingly overriding a virtual function in a base class.)

The needs of both types of clients must be met. A client viewing a class should be able to see all its functionality, regardless of whether a member variable or function was declared in that class or one of its base classes. This is essential in order for clients to have better confidence that an existing class really does meet their requirements.

Also, a client should be able to view the class hierarchy in a suitable (i.e., graphical) fashion.

The Mentor Graphics C++ Browser was in response to these needs. The intent was to create a browsing tool that would fit into the existing environment in a non-intrusive way. We did not want to force users to develop new classes from within the browser. Instead, they are free to use their preferred editing environments for development. Also, we did not want to modify the source code management system already in place for managing versions, doing builds and releases, etc. Other technical objectives were as follows:

- ❑ Selectively view class declarations in one or more header (.h) files
- ❑ Quick response
- ❑ Viewing of all visible member functions/variables of a class, including those defined in base classes
- ❑ Viewing of code/comments related to member functions/variables
- ❑ Graphical display of class hierarchy
- ❑ Allow copying of selected text from the browser into other editors

## 2. Strategy

We abandoned the initial idea of keeping a database of classes and files, in order to avoid the overhead cost of maintaining the integrity of this database. Part of the reason for doing this is that various development groups usually want to browse their development code in the same manner as they would browse the library code. We did not want to spawn browser databases all over the place. Also, code in development is often quite fluid. This would have entailed frequent database update operations.

In our environment, header files are self-contained (i.e., they include any other header files required to compile them). The browser user specifies a header file (or files) of interest. We use the C pre-processor (/usr/lib/cpp) to deal with included files. The resulting stream is read into memory and partially parsed in order to extract the class declarations, which are then stored in a table.

The browser rests on this table of classes. Classes can be added to this table at any time (by specifying additional files to be processed). The user selects a class of interest from this table. At this time, the selected class is parsed, and the results presented to the user. Note that this parsing is quite simple, because we are only parsing a class declaration that is known to be correct (i.e., there is no error handling).

The two stage parsing strategy (one to extract the classes, and one to extract the member variables and functions of a class) is intended to reduce invoke time, by minimizing the amount of processing.

### 3. Capabilities

The C++ class browser's operation may be summarized as follows:

- ❑ From the file(s) specified by the user, the browser generates an alphabetical list of class declarations found. It also generates a tree of class derivations. Additional files can be read in at any time. This causes the alphabetical list to be re-sorted, and the tree to be rebuilt.
- ❑ A user can select classes for parsing either from the tree or from the list. Functions and variables of the selected class are displayed, including visible members from any base class(es).
- ❑ A user can select a particular member variable or function. The browser then shows the actual declaration of the member as it appears in the file, including any comments associated with it.
- ❑ A regular expression based searching capability lets a user search for a class by name, and within a class, search for a specific member function or variable.
- ❑ A user can select a range of text from any of the browser areas, and transfer the selected text into the system paste buffer. From there, it can be pasted to other environments (e.g., editors).

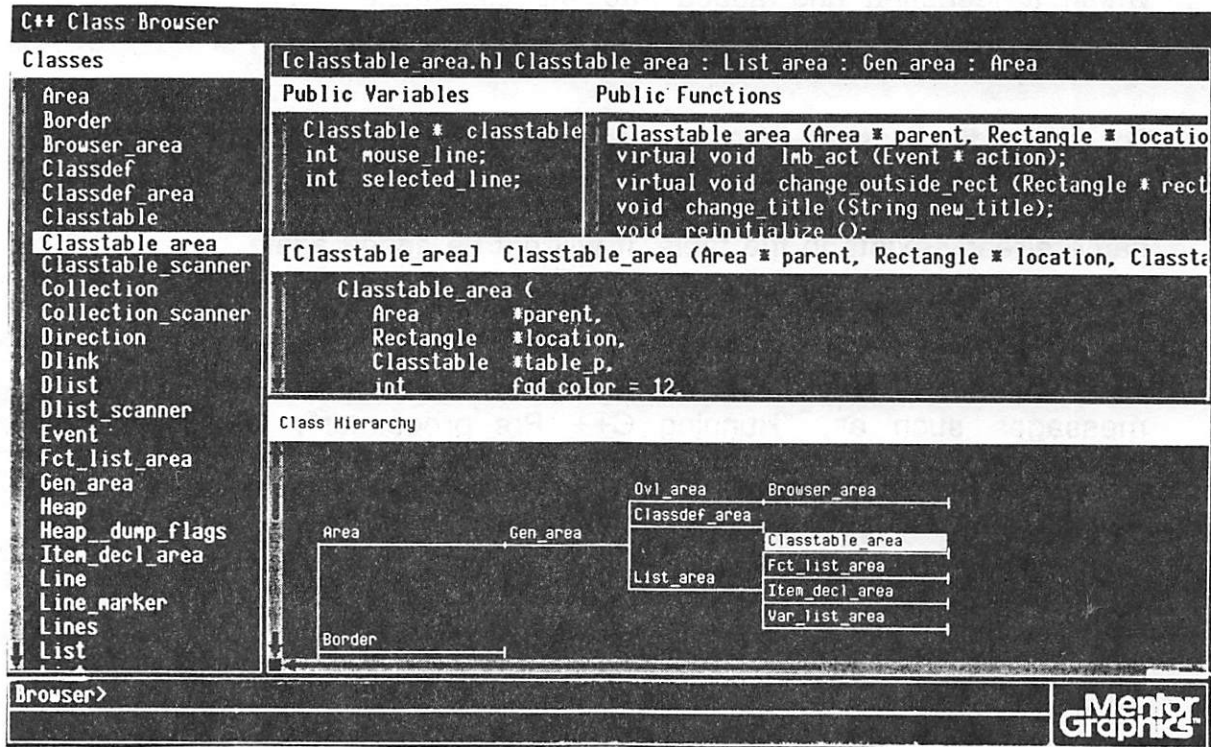


Figure 1: Browser showing all windows

#### 4. User Interface

Interaction with the browser may be broken into two separate activities. The first involves selecting a class of interest. The second involves browsing within the context of a class to examine member variables or functions.

Figure 1 shows a snapshot of the browser. The user can move or grow any or all of the following major windows. (Subwindows of these major windows cannot be moved / grown.)

- for input and status message output
  - Command Line Window

This is where the user can type various browser commands. (These commands may also be invoked via popup menus.)

To read in all the classes from a file, the user types **open <filename>** in the command line. This causes the file to be opened, the classes within it identified and added into the classtable (replacing whatever classes were in it). Simultaneously, a tree representing the class hierarchy is created from the classes in the classtable.

To add to the classes already loaded into the browser, the user types **add <filename>**, which causes the file to be opened and classes within it to be added to the classtable and to the tree. If the class name already exists in the table, it will not be added again.

- o Message window

This window displays information as to the browser's activity with messages such as, "Running C++ Pre-processor", "Parsing Class Definitions", "Unable to Find Pattern [...]", etc.

- for browsing the class hierarchy

- o Class Tree Window

This window shows the class hierarchy of the classtable in graphical form.

- o Class Table Window

This window shows the classes in the classtable in alphabetical order.

A class can be selected by clicking the left mouse button over a class name in either the class table window or the class tree window. The currently selected class is highlighted both in the class tree window and the class table window (Figure 2).

- for browsing the declaration of a particular class

The three windows described below are actually framed by the Class Internals Window. When the user selects a class (either in the class tree window or in the class table window), the class public variables and functions show up in the public variable window and function window (Figure 3). The title of the class internals window shows the name of the class and its derivation hierarchy. It also shows the file in which the class declaration was found.

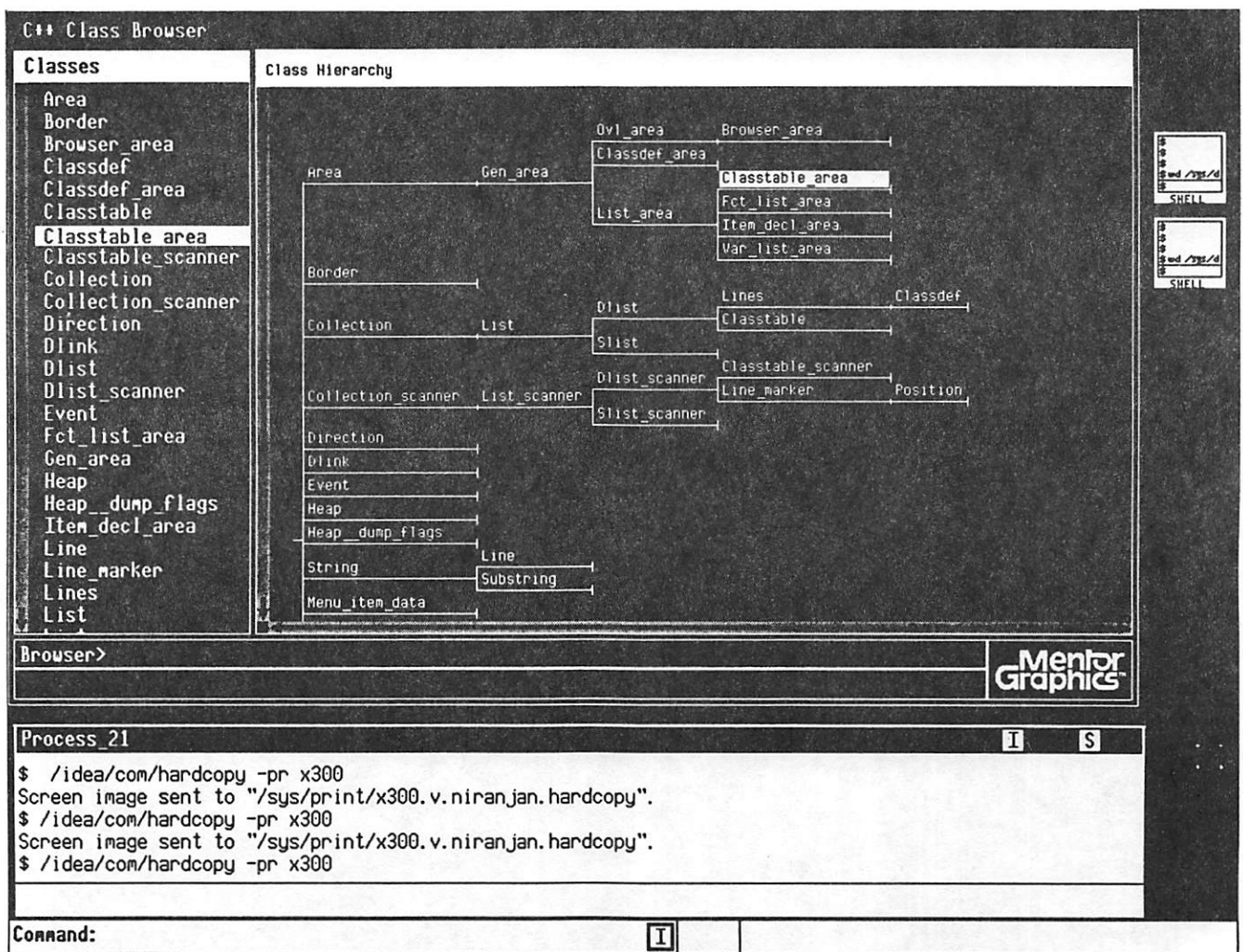


Figure 2: Browser showing class list and class hierarchy

- o Public Variables Window

This window contains a scrollable list of public variables, including visible variables from base classes. The user can select one of these in order to view its actual declaration.

- o Public Functions Window

This window contains a scrollable list of public functions, including visible functions from base classes. The user can select one of these in order to view its actual declaration.

- o Actual Declaration Window

When the user selects either a variable or function to view, the variable or function name highlights itself in the appropriate scrollable list. Also, the

actual declaration of the variable or function in the source file appears in this window (Figure 3). For member functions, the title of this window also contains the full declaration of the function arguments.

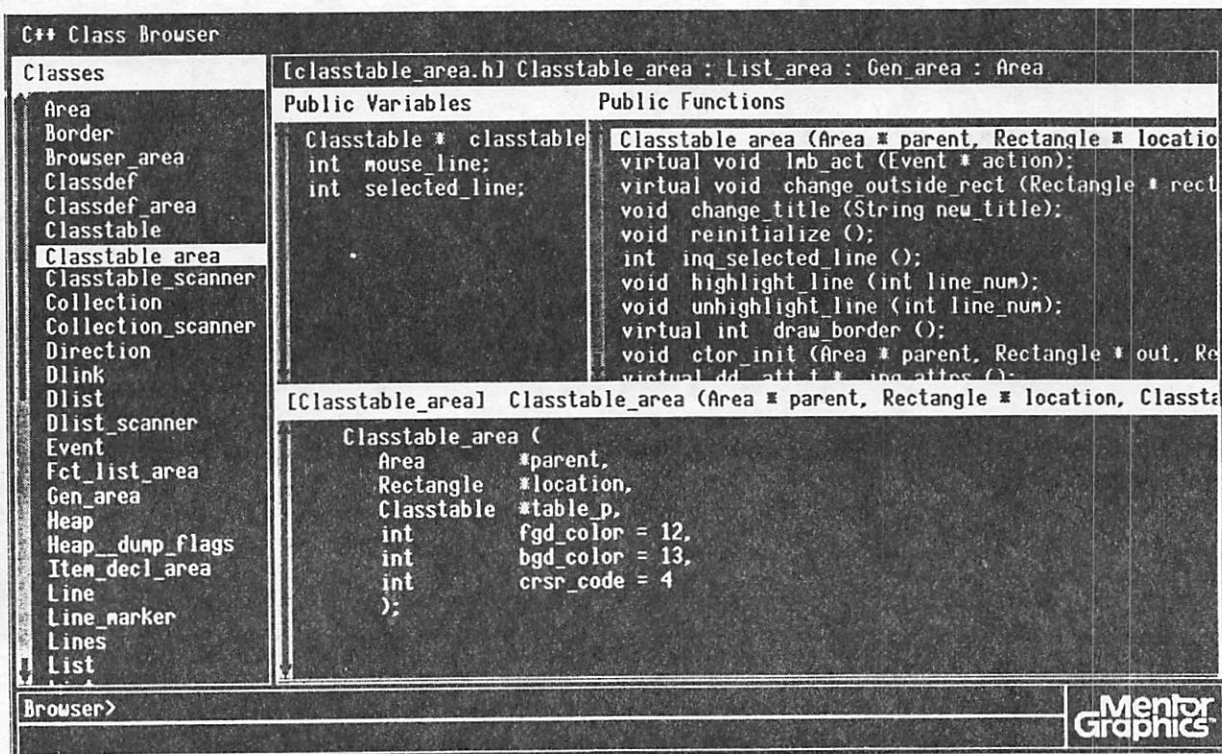


Figure 3: Browser showing class internals

## 5. Implementation

The browser was implemented using the Mentor Graphics C++ class library. Some of the base classes used include **heap** (memory management), **list** (for making collections of things), **string** (dynamic character strings), etc. The user interface is built using **areas**, a Mentor Graphics proprietary user interface management system. The development time for the browser was approximately 1.75 man-months.

The following screenshots taken from the browser depict the class structure used to implement the browser. (In fact, the class derivation tree has proved to be a very valuable documentation tool.

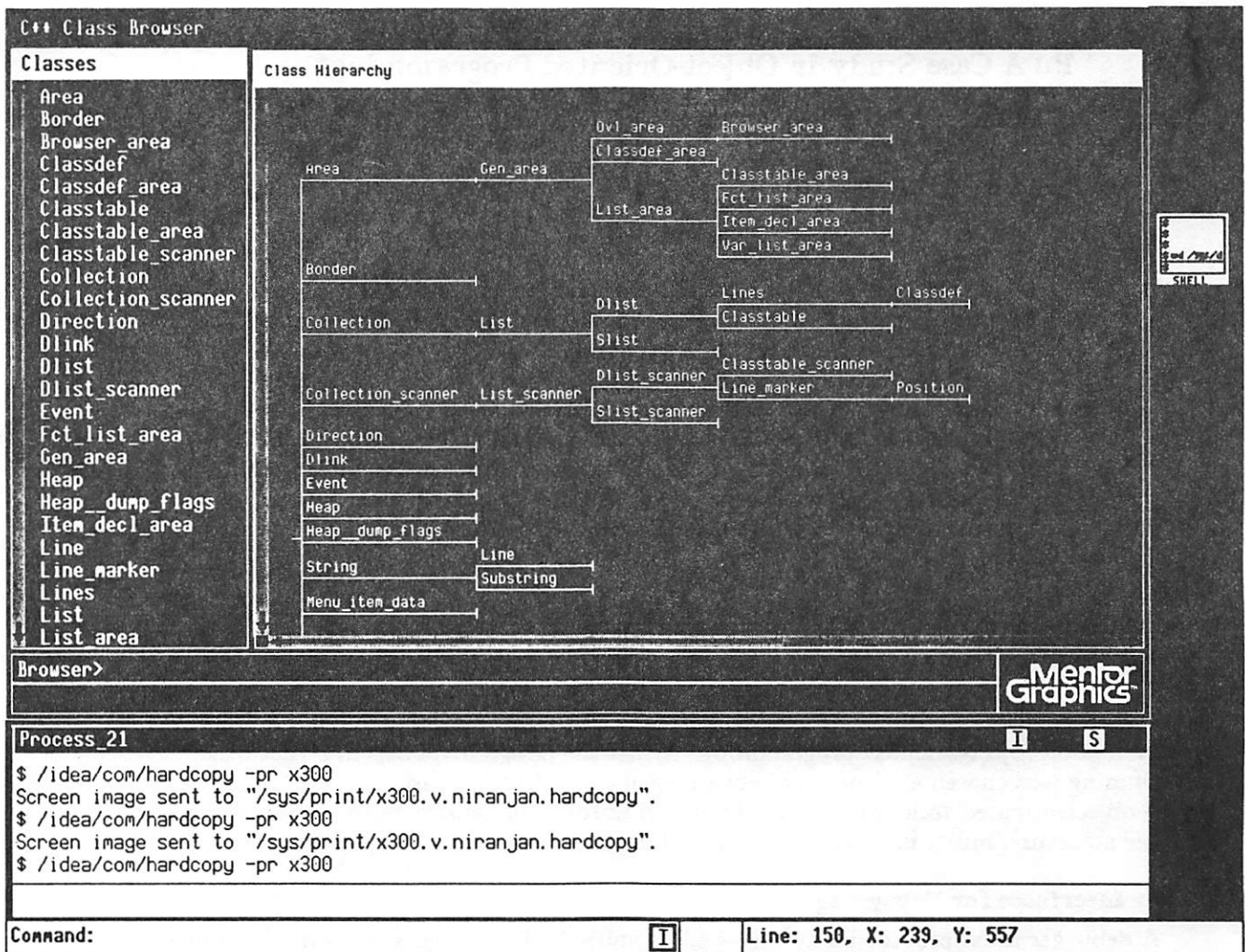


Figure 4: Browser implementation classes

The functions of the various classes can be inferred quite easily. **Classtable** is the internal alphabetical list of **classdefs**, and is displayed in the **classtable area**. The **classtree** area is where the tree depiction of the classtable is displayed. The **classdef area** is where the currently selected class definition data is displayed.

The implementation strategy was to maintain a direct correspondence between the browser data structures and the browser user interface elements.

## 6. Conclusions

The browser has been recently introduced to a few users in Mentor Graphics. It is early to draw any final conclusions about its efficacy, but everyone agrees it is a very useful tool.

The relatively small implementation effort to build the browser once again demonstrates the immense leverage afforded by object oriented programming. C++ has proven to be of great use to the Mentor Graphics software engineering community.

# Pi: A Case Study in Object-Oriented Programming†

*T.A. Cargill*

## ABSTRACT

Pi is a debugger written in C++. This paper explains how object-oriented programming in C++ has influenced Pi's evolution. The motivation for object-oriented programming was to experiment with a browser-like graphical user interface. The first unforeseen benefit was in the symbol table: lazy construction of an abstract syntax-based tree gave a clean interface to the remainder of Pi, with an efficient and robust implementation. Next, though not in the original design, Pi was easily modified to control multiple processes simultaneously. Finally, Pi was extended to control processes executing across multiple heterogeneous target processors.

### 1. Introduction

The subject of this paper is the impact of object-oriented programming on the structure and capabilities of a complicated piece of software — a debugger called Pi. The paper begins with observations about debugging in general and a description of Pi, to motivate the introduction of object-oriented programming. When the design of Pi began, object-oriented programming was chosen as a means of achieving a style of user interface. But the application of object-oriented techniques throughout had unforeseen benefits with respect to symbol table structure, multi-process debugging and target environment independence.

### 2. User Interfaces for Debugging

A debugger must provide many views of its subject. The primary view of the subject program is static: the source text. The primary view of the subject process is dynamic: the callstack, a sequence of activation records. The programmer's insight into the program's behavior comes from merging these views: examining data within the process while controlling its progress through the source text. Debugging purely at the source level may be sufficient for a safe language, correctly implemented, but in practice we also need views of the implementation. Even if none of the subject is written in assembler, it is vital from time to time to consider the process in terms of the instruction stream, registers and uninterpreted memory.

A particular view is not used in isolation; it is related to other views. The programmer moves rapidly among a set of related views. This diversity complicates the user interface. Any attempt to base an interface on a coherent model is confounded by the user's need to switch among so many projections of the underlying subject. For example, the keyboard input

100

could mean "display the context of line 100" of the current source file. It could also mean "evaluate the expression 100" or "display the value of the memory cell at location 100"

† Copyright 1986, Association for Computing Machinery, Inc., reprinted by permission from Sigplan Notices 21:11 November 1986.

and so on. There is no natural interpretation; it depends on the programmer's current focus. If there is some simple model on which a user interface for such a (necessarily) powerful tool could be based, no one has yet found it.

Switching among a set of input modes might help — as long as there is no confusion over what mode is current and how one changes mode. However, experience with just two modes in text editors suggests that a larger number of modes would not work. On the other hand, modeless keyboard languages for debuggers tend to need many qualifiers and options for each command, varying from verbose to cryptic and from pedantic to treacherous. But if they are useful, they are large and complicated. Two recent debuggers use programmability of the user interface to permit the custom definition of keyboard languages for different classes of users. Kraut [4] (with "path expressions") and Dbx [14] (with macros) let new commands be defined in terms of a base language. In contrast, Pi's user interface uses multi-window bitmap graphics, and cannot be extended by the user.

### 3. Pi's User Interface

Pi is primarily an attempt to combine expressive power and ease of use in a graphics interface. The user browses through a network of views, each in its own window with a specialized pop-up command menu and keyboard language. The user selects a window by pointing at it with a mouse or by following a menu or keyboard connection from a related window. All the information within a window is textual, i.e., symbolic and numeric rather than analog or geometric. Moreover, each line of text is also part of the browsing network; it defines its own menu and keyboard interface.

The details of the graphics are not essential, but will help make the discussion more concrete. The display has 100 monochrome pixels per inch. Within Pi, windows may overlap and are positioned explicitly by the user. A proportional scroll bar on the left shows how much of a window's text is visible. The current window has a thick border; the current line is video-inverted. A three-button mouse is used. The left button makes selections: to change the current window, to scroll within it or to select a line. The middle and right buttons raise the pop-up menus associated with the current line and window, respectively. A line of accumulated keyboard characters is displayed in a common space below the other windows. If the current line or window accepts keyboard input, its border flashes at each keystroke. At carriage return a complete line of input is sent to the current recipient.

The following example shows much of the user interface mechanics, but only a few of Pi's features. A more complete example appears in [6]. This trivial C program is taken from [11]:

```
#include <stdio.h>

main()          /* count lines in input */
{
    int c, nl;

    nl = 0;
    while ((c = getchar()) != EOF)
        if (c == '\n')
            ++nl;
    printf("%d\n", nl);
}
```

Assume this program has been compiled and invoked by a command interpreter running elsewhere on the screen. The process reads from the keyboard of its virtual terminal. To bind Pi and the process, the user selects the process from a list of accessible processes in Pi's master window. To control this subject process Pi creates a Process window, for which the user must sweep a rectangle with the mouse. The Process window identifies the process and

shows its state, as in Figure 1. Process 27775 is in a *read* system call, waiting for keyboard input for the call to *getchar()*.

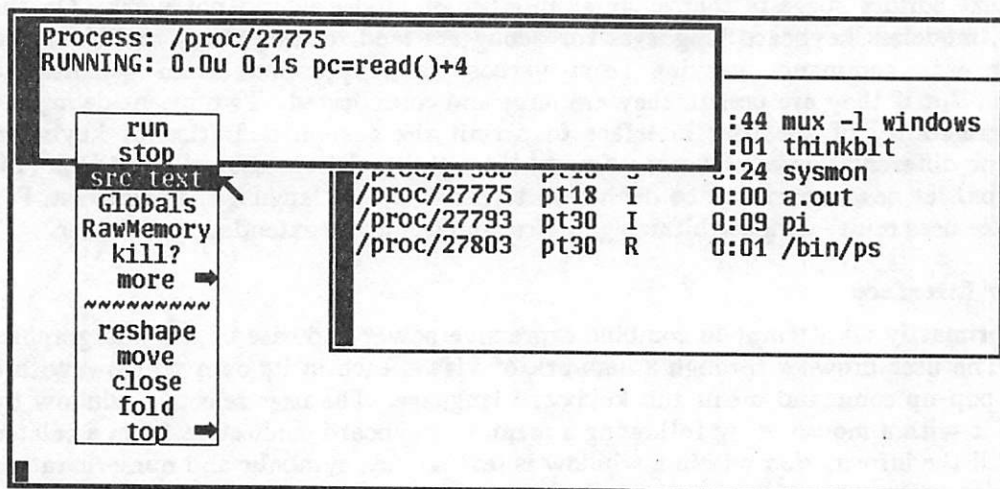


Figure 1. A Process window and its menu.

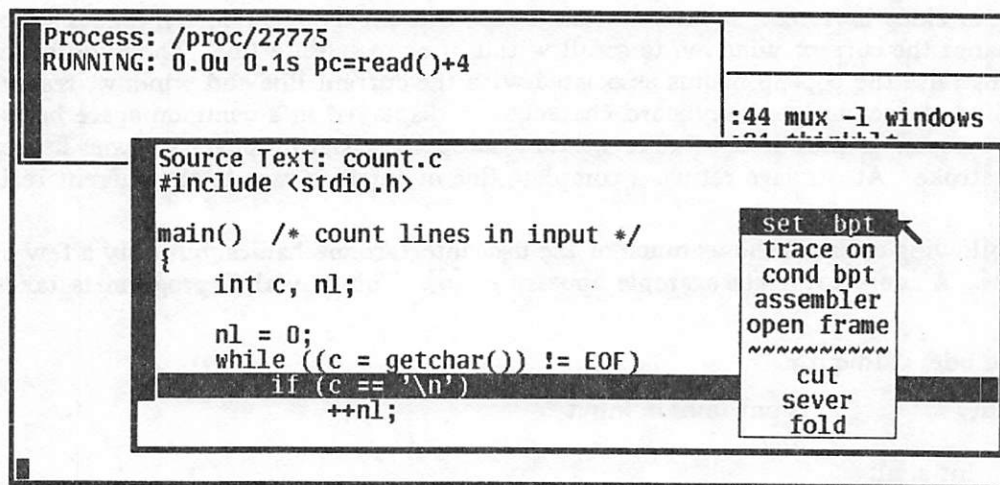


Figure 2. A Source Text window and a source line's menu.

The Process window is the hub of a network of views of the process. Choosing '*src text*' from the Process window's pop-up menu creates a Source Text window for the source file, *count.c*, shown in Figure 2.

A breakpoint is set by pointing at a source line and choosing '*set bpt*' from the line's menu. The presence of the breakpoint is recorded by >>> at the left of the line (Figure 3).

The subject process reaches the breakpoint when the user supplies some keyboard input and *getchar()* returns a character. The Process window announces the change of state and displays a callstack traceback, in this case a degenerate stack of depth 1. Pi shows the

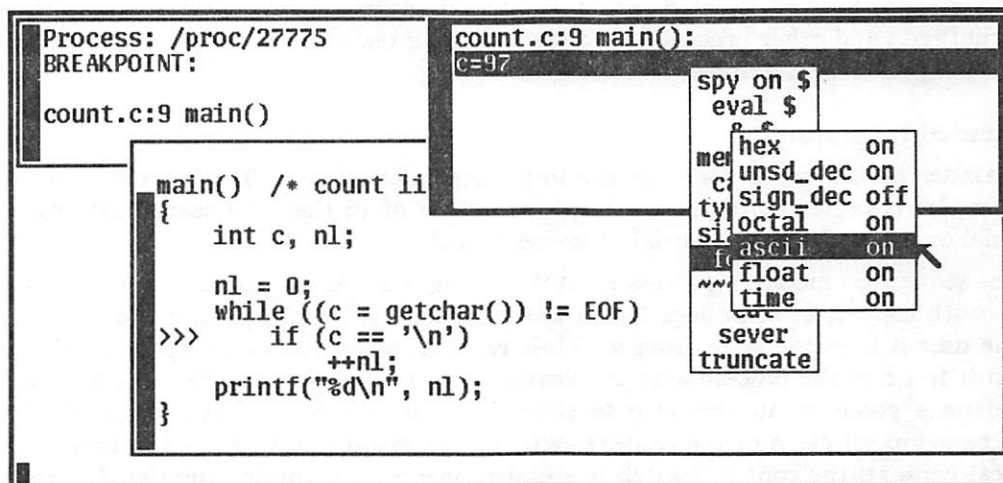


Figure 3. A Frame window and an expression's menu at a breakpoint.

source text context by making the Source Text window current and selecting the source line at which the program stopped. Choosing *'open frame'* from the source line's menu creates a Frame window for evaluating expressions with respect to the activation record associated with that source line.

Expressions in a Frame window are built from menus or the keyboard. Choosing an identifier, say *'c'*, from the Frame's menu of local variables creates and evaluates a simple expression, shown in Figure 3. The variable's type determines the default format in which the value is displayed: an integer in decimal. It makes more sense to display *c*'s value as an ASCII character. Changing the format is one of the operations in the expression's menu. Choosing *'format'* from the expression's menu and *'ascii on'* from a sub-menu of formats re-evaluates the expression and displays it as:

```
c='a'=97
```

In debugging a real program there are likely to be many more windows. (Pi is used daily by dozens of programmers, often on programs of 10,000 lines or more.) The windows shown above may be instantiated arbitrarily often: a Source Text window for each of the program's source files and a Frame window for each of the callstack's activation records. There may be only one instance of each of the other window types: a Frame window bound to global scope rather than an activation record, an Assembler window for controlling the process at the instruction level, a Raw Memory window for manipulating memory as an uninterpreted array of cells, and a Signals window for controlling process exceptions.

The resemblance between Pi and Dbxtool [1] is superficial, even though both debuggers use windows and menus on a bitmap display. Dbxtool is a front-end that translates graphics commands into the command language of a conventional debugger, Dbx. Dbxtool provides a fixed set of windows, one of which, the "Command Window," shows the Dbx commands and responses. To see how this affects the user, consider trying to evaluate expressions with respect to two different activations records at once. In Pi, a window is opened for each activation record and the user moves back and forth with the mouse, evaluating expressions in either window; each window then contains a set of related expressions. In Dbxtool, the user enters a mixture of commands that evaluate expressions and move up or down the callstack; the Command Window then contains a transcript of interleaved context changes and evaluated expressions.

Pi is also unlike the debugger in Smalltalk's "integrated environment," where there is little distinction between the compiler, interpreter, browser and debugger [9]. Smalltalk's tools cooperate through shared data structures and the computer's uniformly defined graphics. On the other hand, Pi is an isolated tool in a "toolkit environment." Pi interacts with graphics, external data and other processes through explicit interfaces. Pi adapts the graphics techniques exemplified by Smalltalk to the toolkit setting.

#### 4. Object-Oriented Programming

The remainder of the paper concerns the implementation of Pi. The object-oriented programming model is better suited to the implementation of Pi than the sequential programming model or the multiple sequential processes model.

Under the sequential model, a process executes a program. At any time the process is in some state, with control at some location in the program. When the process blocks for input from the user it is in the state from which it resumes when the user responds. This makes it difficult to drive the process with the flexibility described above, where the user is allowed to refuse a given menu and leap to some unrelated context. The process must accept either the menu selection or the context switch. The menu selection is a local operation in the local context; the context switch is a global operation involving unrelated parts of the program. This might be achieved by letting the user traverse a network of contexts. But there is then a tradeoff between ease of use and ease of programming: a tree is easy to program, but tedious for the user to traverse; a fully connected network might be easy to use, but calls for every part of the program to be intimately coupled to every other part.

With multiple sequential processes a separate process could be associated with each context. As the user moves around the graphics screen, input is directed to the appropriate process by some agent that maps screen locations to processes. Processes are suspended until the user needs them; they need know only about semantically related processes. As a model this yields a natural architecture, but its implementation is usually very expensive. The overhead for creation and interaction of separate processes might be acceptable for a small number of processes, say one per window. But the desired user interface has each line within each window interacting independently with the user. Many hundreds, even thousands, of processes would be needed. With conventional multi-process techniques the cost is prohibitive.

The object-oriented model lies between these two models. Instead of a collection of processes there is a single process containing a collection of *objects*, each an instance of a type called a *class*. Each object has a copy of the *data members* defined in its class and can execute the *function members* of the class. Unlike a process, an object has no permanent state other than its data. Objects share a universal address space and communicate with one another by invoking function members as procedures. It is feasible to have many thousands of objects. That objects cannot execute concurrently, as processes do by timeslicing a physical processor, is not significant; such concurrency is not required.

The C++ [13] programming language supports this model. C++ is a superset of C with Simula-like classes [3]. The members of a class are data and functions, in private and public sections:

```
class identifier {  
    private_data_declarations  
    private_function_declarations  
public:  
    public_data_declarations  
    public_function_declarations  
};
```

(This grammar generates only a subset of C++ class declarations.) The example of C++ code below defines a class *Const* that privately represents an integer value; the value is rendered as a hexadecimal, decimal or octal character string by three member functions. Similar code is found in *Pi*.

```
class Const {
    int val;      // private data member
public:
    Const(int);   // constructor
    char *hex();  // public
    char *dec();  // function
    char *oct();  // members
};

void Const::Const(int v) // constructor body
{
    val = v;
}

char *Const::hex()       // function member body
{
    build character string
    return pointer to character string;
}

...
```

A member function whose name is the same as that of its class is a *constructor*. If defined, the constructor is invoked to initialize a new object. *Const*'s constructor assigns its argument to the private representation of the value. *Const*'s member functions *hex()*, *dec()* and *oct()* return pointers to ASCII representations of the value. The following client code prints 123 as *0x7B=123=0173*:

```
{
    Const c(123); // c.Const(123) called implicitly
    printf("%s=%s=%s", c.hex(), c.dec(), c.oct());
}
```

Here *Const c(123)* declares an object instantiated on the stack for the lifetime of the block; the argument 123 is passed to the constructor. The object could also be allocated from the free store by means of the operator *new*, in which case the variable's type is pointer-to-*Const*:

```
{
    Const *p;
    p = new Const(123); // from free store
    printf("%s=%s=%s", p->hex(), p->dec(), p->oct());
    delete p;          // return to storage pool
}
```

In general, an executing C++ program consists of a network of objects, referencing one another with pointers. Subject to the encapsulation rules of C++, objects may access one another's member data and invoke member functions. Consider a *Frame* window that evaluates expressions with respect to an activation record. Figure 4 shows two objects: *F*, an instance of class *Frame*, and *E*, an instance of class *Expr*. An arrow represents a pointer; an arrow labeled by a function call indicates that the holder of the pointer may call the member function of the object to which it points.

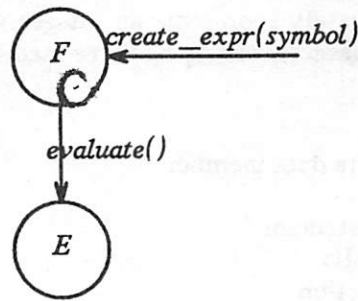


Figure 4. A sub-network of objects.

Here, *create\_expr()* is a member of *Frame* and *evaluate()* is a member of *Expr*:

```

class Frame{
    ...
    void create_expr(Symbol*);
    ...
};

class Expr{
    ...
    ErrMsg evaluate();
    ...
};
  
```

The *Symbol\** argument to *create\_expr* is a pointer to a symbol table object describing a variable in *F*'s activation record's scope, from which the *Frame* creates an *Expr* object *E*. *F* might cause the expression to evaluate itself by calling *E.evaluate()*, and so forth.

The notation in Figure 4 suggests a message-passing model. Indeed, the terminology of object-oriented programming in Smalltalk [9] refers to "messages" between objects. However, the communication is by procedure call.

## 5. Object-Oriented User Interface

A user interface to such a network can be created by letting the user communicate directly through the interfaces that objects present to one another [9]. The user must be able to select an object, see a set of member function calls and select one to be invoked. To receive a member function call from the user an object must describe itself and a set of function calls to the software managing the display.

Figure 5 shows the object network of Figure 4 and a screen image. *F* has associated itself with a window on the screen, and *E* has associated itself with a line in that window. The new "pointers" from images on the screen to objects in the program are shown by dashed arrows. If the user selects *E*'s line, raises the line's menu and selects '*ascii on*', then the result is a call of *E*'s member function:

```
E.reformat(ASCII_ON)
```

Three points of detail should be mentioned. First, the user does not really see the program's internal interfaces, i.e., the identifiers and types of member functions. The text of a menu entry is chosen to support the user's view of the function and need not reflect program's source literally. For example, the user sees '*ascii on*' instead of '*reformat(ASCII\_ON)*'. Second, when invoked, a member function cannot distinguish a call originated directly by the user from a call by another object. There is only one procedure call mechanism. Third, once an object has exposed its image and a set of member function calls to the user, it must be prepared to receive any sequence of calls the user

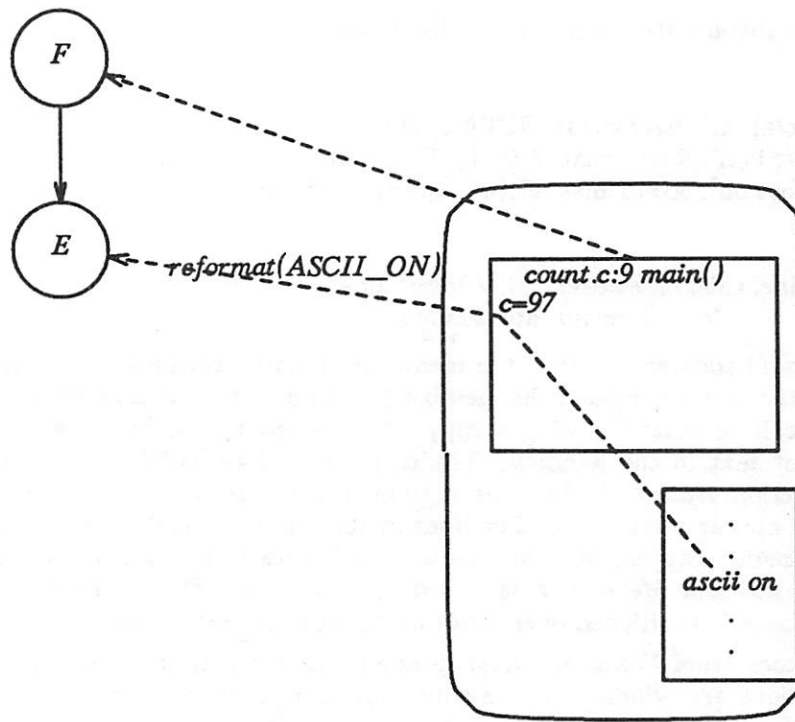


Figure 5. Operations invoked by the user.

chooses to make. To inhibit the user, the object must remove itself or change the set of calls in the menu. Changes in the menu reassure the user about the state of the object: the menu of a line of source text on which the user has already set a breakpoint shows 'clear bpt' instead of 'set bpt'.

The interaction between the user and program is now conducted without a "user interface" in the conventional sense. There is no part of the program responsible for reading, parsing and interpreting a sequence of commands from the user. The user sees a graphical image of the program and the program sees the user as an active participant in its object-oriented world, with implicit software mapping between the two.

## 6. Implementation

The interface between the graphics and the network of objects is a pair of C++ classes: *Window* and *Menu*. These classes in turn interact with a graphics package. To create its window, the *Frame* object executes:

```

w = new Window; // Save the pointer to a new
                // Window in Frame's data member w.
w->bind(this); // Pass a pointer to this (a C++
                // keyword) instance of Frame to
                // the Window.
w->title(description()); // Frame's member function
                        // description() yields
                        // a string like
                        // "count:9 main()".
w->makecurrent(); // To make a newly created window
                // current the user must sweep a
                // rectangle for it.
  
```

To display its value in the window an *Expr* object executes:

```

Menu m; // Instantiate a Menu on the local stack.
.
m.append("octal on", &reformat, OCTAL_ON); // Build
m.append("ascii on", &reformat, ASCII_ON); // the
m.append("float on", &reformat, FLOAT_ON); // menu.
.
w->insert(line, this, m, evaltext()); // Insert line
// with menu into window.

```

Each call to *m.append()* adds an entry to the menu. Each entry consists of the text string the user sees when the menu is raised, the member function to be called and the argument to be passed. The call to *insert()*, using a copy of *w* passed to the *Expr* by the *Frame*, creates a new line of text in the window. The text string, like "c=97", is obtained from *Expr*'s member function *evaltext()*. The *line* argument is a numeric key that determines where this line will appear relative to other lines in the window. Not shown here is how 'ascii on' would be made to appear in a sub-menu, as in Figure 3. Instead of being bound to a window or line of text, one *Menu* may be embedded in another. The embedded sub-menu pops up when the cursor is positioned over a sub-menu indicator in the super-menu.

Note that the code from *Frame* and *Expr* does not reflect the terminal's physical attributes. The abstractions are windows and menus, not coordinates and mouse buttons. Pi has no knowledge of, or control over, details such as the placement of windows, the scrolling of text or the assignment of mouse buttons. Though not used here, the most concrete graphics request is that a line of text be made visible to the user. This means that the graphics and mouse protocol described above is just one of many possible choices; the user's view might be quite different in an implementation for a one-button mouse.

Operations on *Window* objects are translated into a stream of messages passed to a separate process responsible for the real-time graphics. Operations from the user are transmitted back to the main process, received by the package and invoked on the Pi's objects. The two processes execute asynchronously on separate processors: the main process on a Unix† system, "the host," and the real-time graphics process on a Teletype DMD 5620 bitmap terminal, "the terminal" (Figure 6).

† Unix is a trademark of AT&T Bell Laboratories.

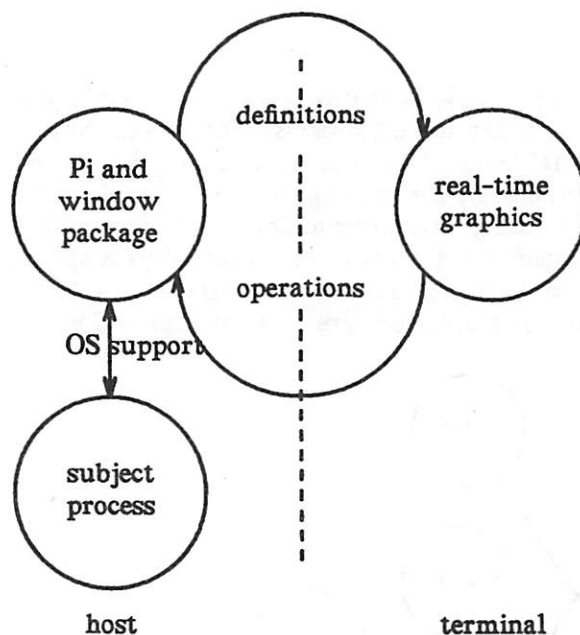


Figure 6. Inter-process communication.

Global control in the graphics program lies in a loop that polls the host and user for commands. The following pseudo-code approximates the loop:

```

for(;;){ // forever
    if( mouse activity ){
        update screen
        if( remote operation selected )
            send message to host
    }
    if( message from host ){
        receive from host
        update screen
    }
    if( real-time clock event ){ // see below
        send message to host
    }
}

```

Global control in the host process lies in a loop in the window package that blocks waiting for messages from the terminal:

```

for(;;){
    read <object.operation.operand> from terminal
    invoke object->operation(operand)
}

```

Usually, an invoked operation in turn causes definitions or redefinitions of objects to be sent to the terminal, to show the user some kind of result. The messages from the host arrive asynchronously with respect to the user's interaction with the graphics. An experienced user need not wait for changes from each operation to be reflected on the screen before invoking another operation. For example, having requested a particular context within a source file to be displayed, the user can select a source line and set a breakpoint on it as soon as that line is visible, even if the host has not finished sending all of the lines needed to fill the window. Some users take advantage of the asynchrony as the natural way to function; others operate as though the communications were half-duplex.

## 7. Pi's Architecture

The debugger is a network of objects as described above. An object of class *Process* is created to take overall control of the subject process. The *Process* object creates two major objects to serve it: a symbol table object of class *SymTab* and a core image access object of class *Core*. The *SymTab* is responsible for reading the symbol table as left by the compiler, assembler and loader and providing that information to the rest of the debugger, as discussed below. The *Core* is responsible for all access to the address space of the subject process and the operating system's control information. Details of the physical processor, operating system and compiler generated code are encapsulated in *Core*.

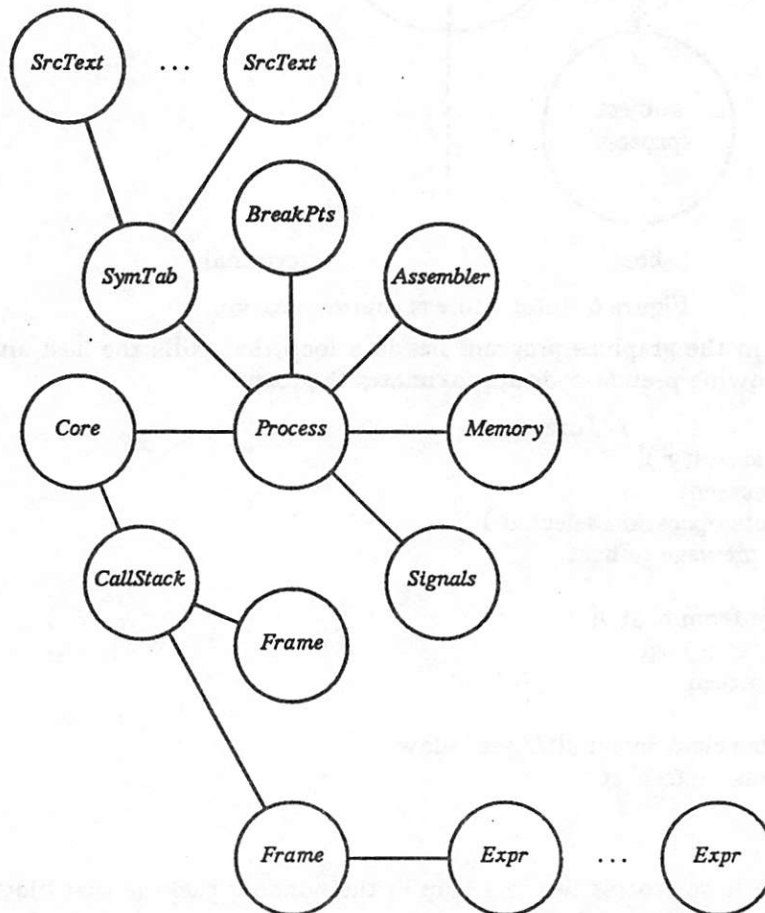


Figure 7. Pi's object network

Of these three classes, only *Process* opens a window, the Process window, from which the user may then open other windows. To display a callstack, the *Process* obtains a *CallStack* object from the *Core* and extracts each activation record it needs as a *Frame* object from the *CallStack*. If a *Frame* is referenced by the user, it opens a Frame window. As the user creates expressions and derives new ones, each expression is an *Expr* object which displays itself as a line in its *Frame*'s window. Figure 7 shows the network. The lines indicate the primary permanent links between objects, but pointers are passed around as needed.

It is the *Process* that monitors the state of subject process. When the subject is running, its state must be polled to see if it reaches a breakpoint or some other exception. The *Process* therefore periodically executes an operation that reads the current state of the subject from the *Core*. This operation re-invokes itself by sending to the terminal a message

requesting that the terminal invoke it after a specified delay. The invocation returned from the terminal is interleaved with, and indistinguishable from, invocations made directly by the user. The *Process* can monitor the state of the subject process while the user asynchronously evaluates expressions, sets breakpoints and so on. This technique creates a few bytes per second of extra host-terminal traffic, since it would be possible to use a clock interrupt on the host instead. However, to guarantee that operations are invoked fairly is much simpler if the only source of operations is a single stream coming from the terminal. Also, the terminal is a better place for real-time programming, both in terms of operating system support and expendable processor resources.

## 8. Symbol Tables

A debugger's needs of its symbol table are similar to those of a compiler, for example, to determine the variables in scope at some point in the program. If the symbol tables prepared by the translator suite reflect the data structures used in the compiler, the debugger is much simplified [5]. If the tables have been "flattened" by an assembler or loader, the debugger suffers the loss of information. Reconstructing an acceptable data structure can be very time-consuming [2]. Working with the flattened tables complicates those parts of the debugger that make non-trivial use the symbol table [7].

For Pi, it seemed likely that versions of the debugger would be used with compilers and assemblers that produced at least two distinct flattened formats. So the first goal was to find a format-independent internal representation that could be built from either format and was well-suited to debugging. Given that the debugger was being written in C++, it seemed a good opportunity to experiment with object-oriented programming. In contrast to the user interface, there was no overall design paradigm guiding the symbol table; the software evolved as different ideas were tried.

>From the outset, the symbol table was a sub-network of objects. Its structure follows the abstract structure of the subject program — a tree. The root of the tree is the *SymTab* object. It reads the file prepared by the loader and builds a tree, as shown in Figure 8. Below the *SymTab* there is a *SrcFile* object for each separately compiled source file that contributed to the program. (There is a one-to-one correspondence between the *SrcFiles* and the *SrcTexts* of Figure 7; a *SrcText* is created and opens a window when the user decides to examine the source text from the corresponding *SrcFile*.) Below each *SrcFile* there is a *Function* for each function defined in that file. Below each *Function* there is a *Block* of arguments and a *Block* of local variables. Each *Block* has a list of *Variables*. Each *Function* also has a list of *Statements*, the source statements in the function. Some symbols can also be accessed directly from a hash table whose entries point into tree's interior.

Like any object, a member of the symbol table can receive operations from the user. For example, the object associated with a line of source text in a Source Window is a *Statement*. To set a breakpoint the user communicates directly with the symbol table.

Though not shown in the figure, each node in the tree has four pointers: to its parent, leftmost child and right and left sibling. These make it easy to traverse the tree. C++'s type inheritance is used to capture the common properties of all symbol nodes in a *base class* called *Symbol* from which other classes of symbols are *derived* :

```

class Symbol {
public:
    Symbol *parent;
    Symbol *leftmost_child;
    Symbol *right_sibling;
    Symbol *left_sibling;
    Address addr;
    char *id;
    virtual char *text();
};

```

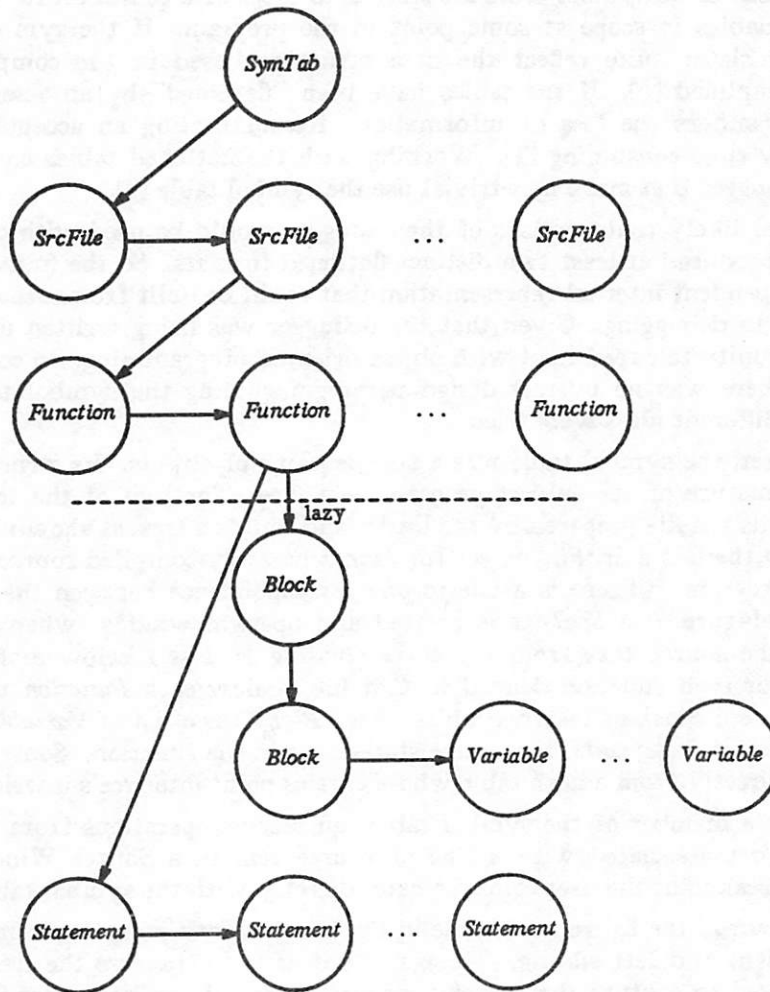


Figure 8. The symbol table hierarchy

The *addr* and *id* data members of *Symbol* record address information and an identifier for each symbol. The function *text()* returns a textual representation of the node, which is just the identifier:

```

char *Symbol::text()
{
    return id;
}

```

There are no instances of class *Symbol* in the tree; each node of the symbol table is of a type derived from *Symbol*. For example:

```
class Variable : public Symbol {
public:
    Storage  storage;
    DataType type;
};
```

As a derived class, *Variable* inherits all the data and functions of *Symbol*; it has two additional data members specific to its needs. In *Symbol*, the function *text()* is declared *virtual*. This means that a derived class may override the base version with its own. *Variable* has no need to do this; it is adequately served by the base version that returns the identifier. However, *Statement* does define its own *text()*:

```
class Statement : public Symbol {
...
    int   line_number;
    char *text();
};
```

Instead of returning the identifier (which is not used by *Statement*), *Statement::text()* walks up the tree to find its *SrcFile* node and returns a string identifying the statement's source file and line number, like:

```
"count.c:9"
```

In general, code traversing the tree to extract textual information does not depend on whether the class of a given node defines its own version of *text()*. For example, *Statement::text()* applies *text()* to a *SrcFile* pointer to obtain the pathname of the source file to embed in the string being built; it does not know whether *SrcFile* has defined its own version of *text()*. The choice of implementation of an operation by the object on which the operation is invoked, rather than the invoker, is common to all object-oriented programming. The technique is used throughout Pi.

## 9. Lazy Symbol Table Construction

If the symbol table as described above were really built it would consume enormous time and space. Early versions of Pi did build it and could only be applied to small programs. Time and space performance was improved dramatically by delaying construction of the subtree of local symbols below each *Function* node until needed — “lazy” construction.

When the debugger picks up a process it scans the flattened symbol table to build the tree only down to the *Function* level — the part of the tree above the dashed line in Figure 8. >From a *Function* the only public access to its sub-tree is through function members. The members of *Function* detect that the sub-tree is missing and call on the *SymTab* to build it before returning a pointer to a requested *Block* or *Statement*. Once the sub-tree has been built it remains; pointers into it may have been passed to, and retained by, other objects.

Lazy construction allows large symbol tables to be presented to the rest of the debugger in a manner that is encapsulated naturally and efficiently. The real-time initialization delay is about one second per thousand lines of source text, on a VAX-11/750† processor. The cost of building a sub-tree on demand is negligible and very few are built. Local tables are needed for only those functions on the callstack or visible in a Source window. It is exceptional to need tables for more than about 15 functions. When Pi, a 500-

† VAX is a trademark of Digital Equipment.

function program, is used to debug itself, it typically builds 1%—5% of the local tables.

This style of lazy table construction could be implemented in any programming language, but reliably only in a language that enforces its data abstraction. In C++, because the only public access from a *Function* to its sub-tree is through member functions, the lazy operation of the symbol table does not depend on cooperation from clients. Very few problems have arisen with this code.

A similar lazy method is used to defer the construction of the tables of user-defined types: the savings are comparable.

## 10. Generators that Traverse the Symbol Table

Clients of the symbol table need to perform various traversals to extract information. For example, a menu built by a *Frame* contains the local variables visible from a function. This might be implemented by a natural traversal of the data structure in Figure 8. However, the symbol table can be better encapsulated by providing a class *VisibleVars*, an instance of which performs such a traversal:

```
class VisibleVars {
...
public:
    VisibleVars(Block*);
    Variable *gen();
};
```

The constructor takes an argument pointing to a *Block*. The only public function, *gen()*, returns a pointer to a different *Variable* on each call, ending with a null pointer. Client code with a pointer to a *Block* from somewhere:

```
Block *b;
```

creates an instance of *VisibleVars* and iterates through the generated variables:

```
{
    VisibleVars vv(b);
    Variable *var;

    while( var = vv.gen() ){
        ...
    }
}
```

This iteration is built on general purpose data abstraction in C++, rather than a built-in iteration primitive [10, 12].

Parts of Pi use nested iteration through the variables visible from a function. Nested iteration arises in the code that warns the user of ambiguity when an identifier occurs more than once in a menu. To determine if an identifier is non-unique, the menu builder searches the identifiers of the variables that its generator will produce later in the iteration by taking a *copy* of the generator in its current state and iterating through the copy:

```

{
  VisibleVars vv(b);
  Variable *v;

  while( v = vv.gen() ){
    ...
    VisibleVars copy(0); // initialized null
    copy = vv;
    Variable *w;
    while( w = copy.gen() ){
      if( strcmp(v->id, w->id) )
        ...
    }
    ...
  }
}

```

## 11. Debugging Multiple Processes

The initial design of Pi did not consider letting the user examine more than a single process at a time. The *Process* class (Figure 7) had been introduced in order to follow the object-oriented paradigm uniformly throughout the program. The intention was to instantiate *Process* only once. Other parts of the debugger were made a little more complicated by this decision because they had to fetch data from the *Process* that might otherwise have been stored in global variables. This version of the debugger did not have a master window or dynamic binding to processes; the subject process was fixed by a command line argument.

Once Pi was working reliably, I realized that with trivial modification it could instantiate an arbitrary set of *Process* objects to examine an arbitrary set of subject processes. The user would need only one copy of the debugger, no matter how many processes were to be examined. It took only a few days to implement. A *Master* object creates a *Process* object for each subject process that the user chooses to examine, as shown in Figure 9. Each *Process* and its sub-network knows nothing of any others that might exist. The user interface package sees no qualitative change — just more objects. The technique whereby a *Process* polls its subject is also unaffected; the delayed invocations of the polling operation from each *Process* are interleaved with one another. None of the problems described in [1] of implementing and using a multi-process debugger have been encountered.

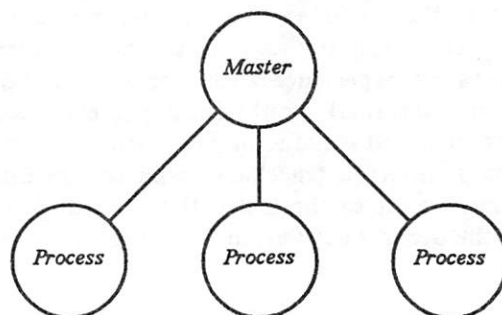


Figure 9. Object network for multi-process debugging.

Multiple process debugging could also have been achieved by instantiating multiple debugger processes rather than multiple *Process* objects within a single debugger. The advantage of instantiating only a single debugger is the reduced overhead for both the user and the computers. With only one instance of the debugger the user has less to manage on the screen. When multiple processes are debugged the set of debugging windows are the

same whether they are together in one debugger's window or in separate instances of the debugger. But when the user wants to treat the debugging environment as a whole, it is better to deal with a single tool. For example, removing a single debugger is simpler than removing a set of debuggers. Less machine resources on the host and terminal are required to execute only a single process in each. In special circumstances (such as debugging a debugger), multiple instances of the debugger are needed.

## 12. Multiple Target Environments

The initial design did anticipate versions of Pi that would operate in different target environments. Those parts of the debugger dependent on the target processor were encapsulated in the *Core* and *Assembler* classes, those dependent on the operating system in *Core*, and those dependent on the external symbol table format in *SymTab*. The original version of Pi was for a VAX processor running the Eighth Edition of the Unix system. The intention was to tailor versions to different target environments as the need arose. The first demand was for a version to debug processes in the DMD 5620 bitmap terminal: an AT&T WE32000 processor running a virtual terminal multiplexor called Mux.

As a further experiment with object-oriented programming, I decided to build both of these versions as a single program — so that one instance of Pi could simultaneously examine processes in both target environments. For each target-dependent class there must be a base class, with a derived class for each target environment. Everywhere a target-dependent object is instantiated it must be of the appropriate derived class, but its target-independent clients need not know from which derivation. The classes derived from *Core* are *HostCore* and *TermCore*, for the host and terminal, respectively. The class hierarchy for *Core* is shown in Figure 10.

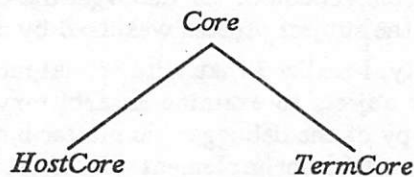


Figure 10. Class hierarchy for *Core*.

The main process of Pi is still a process in the timesharing host. To access memory in the terminal, an instance of *TermCore* (executing on the host) communicates with an additional agent process in the terminal. (The agent process need not be in the same terminal as the real-time graphics process, but usually it is.) This communication is based on remote procedure calls from the host to the terminal. The debugger is now three processes altogether: the host process, the real-time graphics process and the terminal access agent, as shown in Figure 11. Implementation experience with a previous debugger [7] indicated that bandwidth between the host and terminal would limit performance. This influenced the design of the host/terminal protocol and satisfactory performance was achieved. For example, when a *TermCore* requests a callstack traceback from the terminal, the terminal computes the callstack locally, compares it to the last callstack sent to the host and transmits the difference — usually only the deepest activation record has changed.

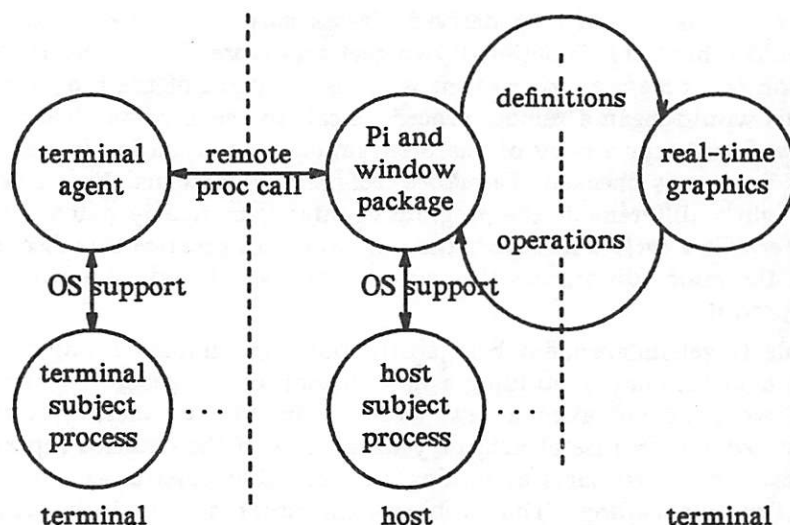


Figure 11. Pi's three processes with two subject processes.

How *Core* finds the value of the subject's program counter is a simple example of inheritance and virtual functions at work. Consider part of the definition of *Core*:

```
class Core {
...
public:
virtual int pc_index(); // register number
                        // for program counter
virtual long reg_save(int r); // address at which
                        // register r saved
virtual long peek_long(long a); // fetch value from
                        // memory at address
virtual long pc(); // fetch value of program counter
...
};
```

The code here has been somewhat simplified to eliminate irrelevant complications; for example, it doesn't handle errors. *Core::pc()* is target-independent, though it calls the target-dependent functions *pc\_index()*, *reg\_save()*, and *peek\_long()* to obtain its result:

```
long Core::pc()
{
    return peek_long( reg_save( pc_index() ) );
}
```

*pc\_index()*, *reg\_save()* and *peek\_long()* must be implemented for each of *HostCore* and *TermCore*. For example, the program counter is register 15 on the VAX:

```
int HostCore::pc_index()
{
    return 15;
}
```

*reg\_save()* and *peek\_long()* have the target-dependent code to find the location at which an arbitrary register has been saved and read the contents of an arbitrary memory location, respectively.

Note that *Core::pc()* is virtual; the derived classes *may* also redefine it. So, even though *TermCore* could inherit this functionally correct *pc()* from *Core*, it has its own version. The base version reads memory every time it needs the value of the program counter. For the terminal, this would mean a remote procedure call to the terminal every time. As an optimization, *TermCore* keeps a copy of the program counter, updating it each time the state of the subject process is checked; *TermCore::pc()* simply returns this cached value. The semantics are slightly different: if the program counter is manually patched while the program is halted, *TermCore::pc()* will report the old value. In practice this discrepancy is less significant than the minor differences that arise from operating system idiosyncrasies. No user has ever noticed it.

Finding suitable target-independent base abstractions and implementing the derived classes took several months; simply building a new version of Pi specifically for the new target environment would have taken a few weeks. The *SymTab* class was relatively straightforward, but tedious because of arbitrary differences in the detailed representation of the symbol tables. Two hard parts of finding an acceptable inheritance for *Core* were byte ordering and function calling. The problems encountered with byte ordering are instructive — the original scheme did not work on either machine, for reasons that no amount of forethought (by me) would have revealed. The scheme is to read memory from the subject process and create objects from which various types of data (byte, short, long, float, double) can be extracted later by clients of *Core*. The VAX version failed when it tried to set up arbitrary bit patterns as candidates for extraction as floating point values; the operand of a floating move instruction must be a valid floating point number, of which 1 in 256 bit patterns is not. The WE32000 version did not work because the processor does not use the same byte ordering for code and data fetches; a multi-byte constant embedded in code is not the same bit pattern as that constant in data. Neither of these problems was hard to fix, but they indicate the difficulty of finding machine-independent abstractions for hardware. Harder was the interface through which the expression evaluator calls a function in the subject. The mechanisms for the host and terminal are quite different. For the host architecture, the debugger arranges that the subject process execute the function using the user's stack; in the terminal the function is executed directly by the debugger's agent process in the terminal on its own stack. The operation is broken down into a series of steps, each performed by a member of the respective derived *Core*, such that *Expr* can detect no difference. The five steps supplied by *Core* are: save context, allocate argument area, call function, determine location of returned result, restore context.

A further derivation from *HostCore* has been added for examining core dumps from the Unix kernel on the VAX. *KernelCore* differs very little from *HostCore*; the major change is that memory fetches must be mapped through the kernel's page tables. A derivation of *Core* for S-Net, a multi-processor computer based on the Motorola MC68000, is being implemented at the time of writing. The current *Core* hierarchy is shown in Figure 12.

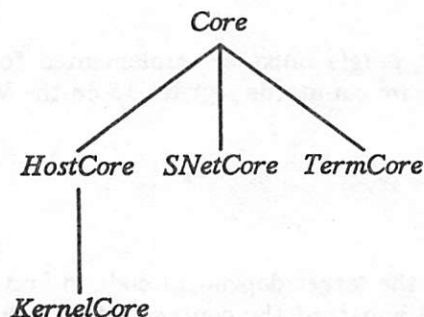


Figure 12. Current *Core* hierarchy.

A single debugger that handles multiple target environments has been a success for both the users and the implementer. The user is guaranteed to see the same interface when debugging in all environments. When changes are made to target-independent parts of Pi they are usually only tested for one target before being installed — they almost always work correctly for the others. This was true of adding a trace history of breakpoints, for example. More complicated changes, involving target-dependent parts, take some time before a clean compilation can be achieved, because several derived classes must be changed consistently. It often takes days to get an error-free compilation. It is frustrating to be unable to test new code for machine X because the code for machine Y is out of date and cannot compile. The discipline introduced is that thought must be given to all target environments simultaneously; this results in earlier exposure of target environment inconsistencies.

### 13. Deficiencies in C++

Though indispensable in the construction of Pi, C++ is deficient in two respects. First, derived classes may inherit from only a single base class. Second, the benefits provided by classes come at the expense of considerable compilation overhead.

Though each class derived from *Core* is a single step from its parent in the type hierarchy, the step embodies several independent changes: the processor, the operating system and the compiler. If Pi had to support all four target environments possible with operating systems A and B on processors P and Q, the class hierarchy would be that of Figure 13. As a result, each derived class would contain target-dependent code replicated in two others.

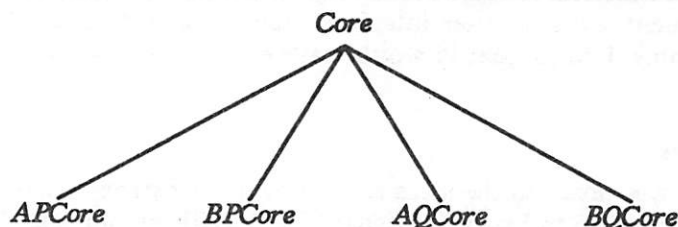


Figure 13. Multiplicity of Derived Classes.

This is a situation in which *multiple inheritance* could be used to eliminate replication. Under multiple inheritance a derived class may inherit from more than one base class. The derivation graph could be as shown in Figure 14. Target-dependent code would be confined to a single appearance in one of the base classes: *ACore*, *BCore*, *PCore* and *QCore*. The derived classes would need no additional code.

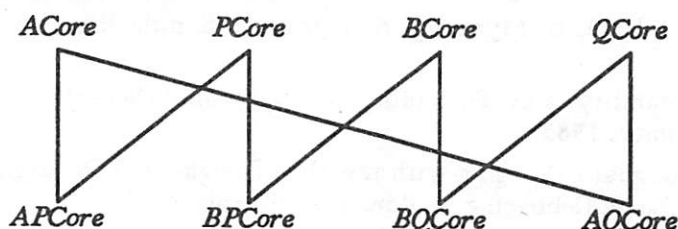


Figure 14. Multiple Inheritance.

No pair of target environments have yet shared a common component, but time will certainly change that. Since C++ does not provide multiple inheritance, some other means of factoring the program must be found to avoid duplicating code. Of course, the success of multiple inheritance cannot be guaranteed without practical experience, but it is certainly worth pursuing.

A more severe problem that has been encountered is the cost of recompilation triggered by the modification of class declarations. The declaration of a class, *X*, places both the public and private components of its interface in a single syntactic unit. The declaration is usually stored in a "header file," *X.h*. Two compilation problems arise with respect to *clients* of *X*, that is, other classes that depend only on *X*'s public interface. Before processing the source text of a client the compiler must read the header file, *X.h*. It therefore reads both the public and private declarations in *X*, even though the client's source is denied reference to the private declarations. (At the implementation level, the client might depend on this private information: to generate client code, the compiler needs to know the *size* of the private data in *X*, if an instance of *X* appears in a client's stack frame.) If the private declarations in *X.h* in turn depend on other header files, they must also be included, and so on. Compilation of the client therefore depends on many header files, even though the client does not need the information from those header files. This makes client compilation more expensive, because of the additional header files that must be processed. Moreover, using conventional Make [8] dependencies, client source is frequently recompiled after the modification of private declarations in classes unreferenced by the client. The subterfuge that partially overcomes this problem is unworthy of description.

#### 14. Conclusion

Object-oriented programming in C++ has worked very well in Pi. Pi's ability to examine multiple processes over multiple target environments follows from the object-oriented model and class inheritance mechanism used in the implementation. At the outset the goal was to experiment with the user interface. Had an object-oriented programming language not been available, I doubt that Pi would have evolved beyond experiments at that level.

#### 15. Acknowledgements

The success of Pi owes much to the ideas and software of Bjarne Stroustrup, Tom Kilian and Rob Pike. Thanks also to Brian Kernighan, Doug McIlroy and John Linderman for their comments on drafts of this paper.

#### 16. References

1. E. Adams, S.S. Muchnick, 'Dbxtool — A Window-based Symbolic Debugger for Sun Workstations', Proceedings Summer USENIX Conference, Portland, Oregon, July, 1985.
2. B. Beander, 'VAX DEBUG: an Interactive, Symbolic, Multilingual Debugger', Proceedings of Symposium on High Level Debugging, Asilomar, California, 1983.
3. G.M. Birtwistle, O-J Dahl, B. Myrhaug, K. Nygaard, 'Simula Begin', Chartwell-Brat, 1980.
4. B. Bruegge, 'Adaptability and Portability of Symbolic Debuggers', Ph.D. Thesis, Carnegie-Mellon University, 1985.
5. J.R. Cardell, 'Multilingual Debugging with the SWAT High-level Debugger', Proceedings of Symposium on High Level Debugging, Asilomar, California, 1983.
6. T.A. Cargill, 'The Feel of Pi', Proceedings Winter USENIX Meeting, Denver, January, 1986.
7. T.A. Cargill, 'Implementation of the Blit Debugger', Software — Practice and Experience, 15, pp. 153-168, 1985.
8. S.I. Feldman, 'Make — a Program for Maintaining Computer Program', Software — Practice and Experience, 9(5), 1979.
9. A. Goldberg, 'Smalltalk-80 The Interactive Programming Environment', Addison-Wesley, 1984.

10. R.E. Griswold, M.T. Griswold, 'The Icon Programming Language', Prentice-Hall, 1983.
11. B.W. Kernighan, D.M Ritchie, 'The C Programming Language', Prentice-Hall, 1978.
12. M. Shaw, W.A. Wulf, R.L. London, 'Abstraction and Verification in Alphard: Iteration and Generators', Communications of the ACM, August 1977.
13. B. Stroustrup, 'The C++ Programming Language', Addison-Wesley, 1986.
14. 'Unix Programmers Manual BSD 4.2', University of California, 1984.

10. B.R. Grew, "The C++ Programming Language", Addison-Wesley, Reading, MA, 1985.
11. A.W. Hughes, "The C++ Programming Language", Addison-Wesley, Reading, MA, 1985.
12. A.W. Hughes, "The C++ Programming Language", Addison-Wesley, Reading, MA, 1985.
13. A.W. Hughes, "The C++ Programming Language", Addison-Wesley, Reading, MA, 1985.
14. A.W. Hughes, "The C++ Programming Language", Addison-Wesley, Reading, MA, 1985.
15. A.W. Hughes, "The C++ Programming Language", Addison-Wesley, Reading, MA, 1985.
16. A.W. Hughes, "The C++ Programming Language", Addison-Wesley, Reading, MA, 1985.
17. A.W. Hughes, "The C++ Programming Language", Addison-Wesley, Reading, MA, 1985.
18. A.W. Hughes, "The C++ Programming Language", Addison-Wesley, Reading, MA, 1985.
19. A.W. Hughes, "The C++ Programming Language", Addison-Wesley, Reading, MA, 1985.
20. A.W. Hughes, "The C++ Programming Language", Addison-Wesley, Reading, MA, 1985.

## CLAM – an Open System for Graphical User Interfaces

*Lisa A. Call*  
*[lisa@grilled.wisc.edu](mailto:lisa@grilled.wisc.edu)*

*David L. Cohrs*  
*[dave@romano.wisc.edu](mailto:dave@romano.wisc.edu)*

*Barton P. Miller*  
*[bart@asiago.wisc.edu](mailto:bart@asiago.wisc.edu)*

Computer Sciences Department  
University of Wisconsin – Madison  
1210 W. Dayton Street  
Madison, Wisconsin 53706  
(608) 262-1204

### ABSTRACT

CLAM is an object-oriented system designed to support the building of extensible graphical user interfaces. CLAM provides a basic windowing environment with the ability to extend its functions using dynamically loaded C++ classes. The dynamically loaded classes allow for performance tuning (by transparently loading the class in either the client or the CLAM server) and for sharing of new functions.

In addition to the traditionally layering of output abstractions, CLAM allows the programmer to easily layer input abstractions. The input functions include providing distributed upward calls through the layers, and light weight processes to support asynchronous input.

CLAM is currently running under 4.3BSD UNIX on a MicroVax-II workstation.

## 1. Introduction

CLAM is an open system for graphical user interfaces. The overall goals for CLAM are to supply a flexible and extensible environment for the development and support of parallel and distributed programming applications, and to provide tools for debugging, performance monitoring and specifying such applications. CLAM provides a framework on which a consistent user interface can be made available to the various applications. This interface is based on providing windows on a bitmapped display and various devices for user input, such as a pointing device and a keyboard. CLAM's extensibility gives us the ability to provide an efficient common user interface. The current implementation is on 4.3BSD UNIX [1].

CLAM uses the framework supported by object-oriented data abstractions, such as those in Smalltalk [2] and C++ [3]. Through the use of inheritance and layered classes we have the ability to build flexible abstractions. The layered approach provides us with a user interface that can be easily extended by adding or replacing specific layers provided by the environment. An object-oriented system which provides sharing in a distributed environment, while keeping the concept of layered classes, is especially useful in designing a graphical user interface. The object-oriented environment of Emerald [4] has a different philosophy of sharing, not based on inheritance, but does provide sharing and abstraction in a distributed environment.

Flamingo[5] is another object-oriented system designed for user interface management. It also employs remote procedure calls (called remote method invocation in Flamingo). CLAM uses a much different approach for handling input events, supporting extensions to the basic system, and providing protection. The emphasis in Flamingo is towards supporting multiple display managers.

The overall structure of CLAM is based on the client/server model. Other interface managers, such as NeWS [6] and the X Window System [7] use this model. This model allows us to place low level, commonly shared routines into one package, providing a consistent interface to all clients. Flexibility in CLAM is achieved by allowing each client to specify extensions of these low level functions tailored to their specific applications. This model allows the applications to share all of the abstractions presented by the server.

We chose a less restrictive model than employed by some existing server models. X, for example, provides a fixed set of functions in the server. This can lead to a server whose design must be continually

expanded to match changes in its application community. If an application requires some extension to these functions, for example, dragging a user-defined object around inside of a window with the mouse, input events must be sent from the server to the client, which must update the position of the object and send commands back to the server to update the display. This communication can result in lower performance. We would like an extensible environment that would allow clients to dynamically create objects, layers, and extensions as needed, and, for improved performance, to load these classes into the server itself. This would lead to an environment specifically tailored to the needs of each client but with the advantages of having the objects and operations existing in the server. Using this feature we can tune the performance of the system by careful separation of the application into the client and server parts.

### 1.1. Design Goals

We want the extensibility that is provided by a dynamic environment and to be able to specify these extensions in a natural way. NeWS provides extensibility by communicating with its clients using an extended version of Postscript[8], which allows the definition of new functions at execution time. Unfortunately, NeWS requires the programmer to manipulate the display using a language different from the one in which they write their applications. In addition, the interpreted approach used in NeWS may have lower performance than precompiled functions. To address these issues, we base our environment on C++, an object-oriented high level programming language, and provide dynamical loading of precompiled classes. The programmer has the choice of either statically linking the classes to the application program (as is usually done) or dynamically loading the class into the server.

The interface manager should provide support for both input and output. Output is a well understood problem, and all existing interface managers handle it well. Input, especially in an layered, object-oriented environment, is harder to support in a consistent manner. The approach taken in Swift [9], called *upcalls*, allows low level abstractions to call higher levels in an environment contained within a single address space. We extend the upcall method to *distributed upcalls*, which allows the low level objects to make upcalls that can cross the process boundary back to a client. We would like to provide for the maximum amount of concurrency between applications for both input and display operations. To achieve this goal, the CLAM server provides light-weight processes to the applications. NeWS also provides light-weight processes to its clients. Light-weight processes can increase the concurrency possible in updating the

display. They also simplify passing asynchronous input events upwards through the levels of abstraction.

There are several other goals for the CLAM design. First, we want to provide a higher level of sharing of abstractions than is currently available. Second, we want to provide protection in the interface manager, both between clients and from intruders. Last, we want to be able to tune performance by distributing an application between the clients and the server.

The CLAM design addresses these problems. It provides a dynamic loading facility based on the high-level, object-oriented language, C++. The client/server model is employed for the interface manager, but the operations provided by the server are not fixed at compile time; rather, the server defines a set of primitive operations and is extensible both in the number and types of objects it can support. This model, in an object-oriented environment, allows us to provide sharing and protection, and flexibility in performance tuning. Through the use of distributed upcalls, CLAM provides the same level of support for input management as is commonly available for output.

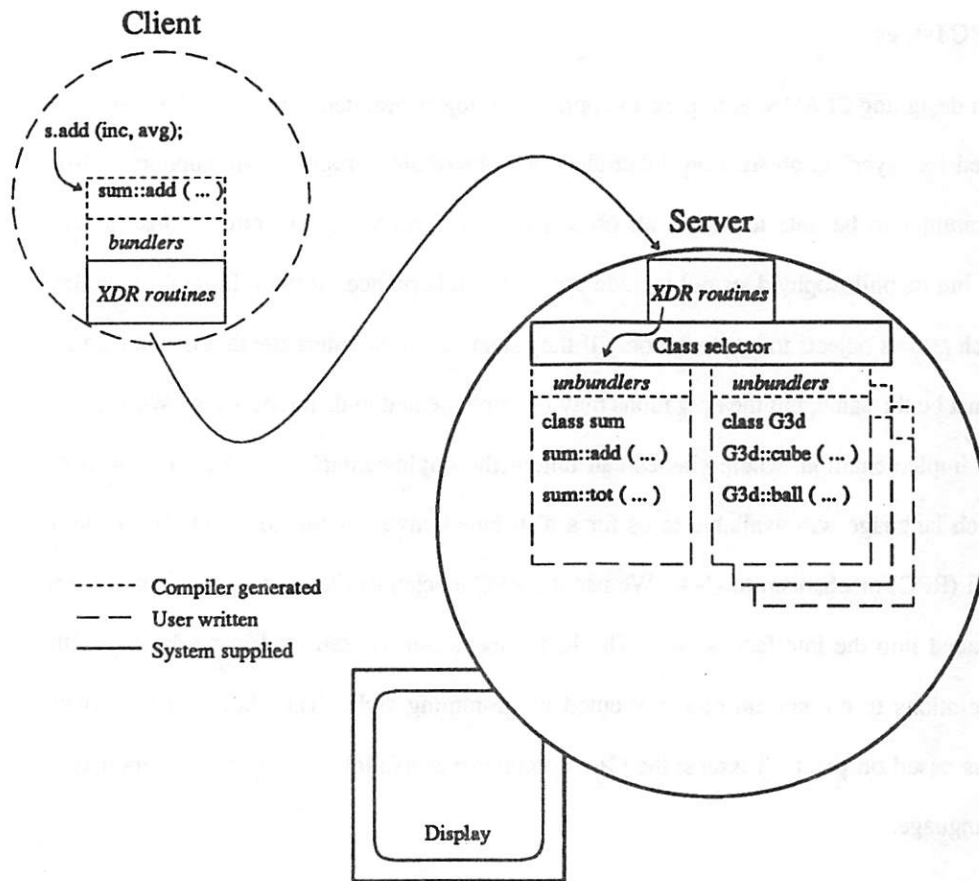


Figure 1: Overview of CLAM System

## 1.2. Structural Overview

Figure 1 shows an overview of the CLAM system. Clients access the basic functions of CLAM by using remote procedure calls. The user writes these remote procedures (indicated by the dashed lines) using the class structure of C++, and can load them into the CLAM server. Our C++ compiler automatically generates the RPC stub routines (shown in the dotted lines).

Our discussion of the design and implementation of CLAM is as follows. First, we address the language and remote access issues. Next, we discuss CLAM's dynamic loading and binding mechanism. This includes a discussion of techniques used for version control and protection. Third, we discuss our approach to the asynchronous input problem, including support for threads of control that cross address

space boundaries. Finally, we give our conclusions and the current status of the project.

## 2. Language and RPC Issues

The first step in designing CLAM was to pick an appropriate object-oriented language. We wanted a language that provided for layering, abstraction, inheritance and distributed programming support. Also, we wanted the programmer to be able to access all objects in a uniform way. Emerald is close to the language we wanted, but its philosophy does not include the idea of inheritance. Rather, Emerald includes *type conformity*, which allows objects to be shared only if they share the same interface; that is, the parameters to operations must be the same, but the operations may be implemented in different ways. We wanted inheritance based on implementation, where classes can inherit the implementation of an operation from other classes. No such language was available to us for a distributed environment, so we chose to add a remote procedure call (RPC) mechanism to C++. We use the RPC mechanism to access classes that have been dynamically loaded into the interface server. The RPC mechanism is restricted to work only with classes and class operations to enforce an object-oriented programming style. The choice of C++ over similar languages was based on practical issues: the C++ compiler was available to us and is a good systems programming language.

An important goal in our design was to integrate, as much as possible, the RPC mechanism into the existing programming language. In some existing languages, for example, in Cedar Mesa[10], a separate language is used by a *stub generator* to describe the parameters to remote procedures and the manner in which they should be passed to the remote procedure. We felt that this duplication of effort was unnecessary. To add RPC to C++, we extended the specification of formal parameters and return values in the language and require a few programming style conventions not normally required in C++. A stub language is not needed and, in most cases, modifications do not need to be made to existing class definitions to use the RPC mechanism. The modified C++ compiler will generate, given only the class specification and the class functions, object code to operate on objects on this class as well as the client and server stubs for parameter passing.

## 2.1. Parameter Bundlers

Each procedure that is called remotely (from another process) requires additional code to package parameters, ship them to the remote site, unpackage the parameters, and make a local call to the procedure. Output parameters are handled in a similar manner. Procedure return values are handled the same as are output parameters.

Given a formal procedure declaration, our modified C++ compiler automatically generates a *bundler* for each of the parameters. The bundler packages the parameters and uses SUN's External Data Representation[11] to pass them from the client to the server or vice versa, in a machine independent manner. In many cases, the compiler can decide on how to package a parameter. Parameters that contain no pointer references are packaged as simple data items or groups of simple items. Parameters that contain pointers are more complex. One alternative is to have the compiler (or RPC stub generator) form the transitive closure by following all contained pointers, and packaging all data in the closure. While this is conceptually the cleanest approach, this approach has several problems relating to system performance.

First, the mechanism to generate the closure and to follow the pointers to package them is complex, and this complexity can add cost to all bundling activities. Second, there are cases where generating the closure may not be the desired action. For example, to pass a single element from a linked list might cause a large part (or even all) of the list to be packaged. While this is logically correct, the performance cost could be large.

By default, our compiler does not follow pointers in data structures but passes them as raw data. These pointers are unusable to the server but usable if passed back to the client. If the programmer wishes to pass parameters in some way other than the default, they may supply a bundler for this special purpose. This escape mechanism allows the compiler to automatically bundle parameters in the simple cases, but allows C++ parameter semantics to be preserved through manual intervention.

We provide a library of common user specified bundlers. The most common example is for character strings. In UNIX, character strings are most often represented as a sequence of characters terminated by a null (zero) value character. The string is accessed by using a character pointer. There is a user bundler that understands this format and correctly passes strings to remote procedures. Figure 2 shows an example of a user bundler declaration. The class *sum* is used to keep a running total of values. The

sum::makelabel procedure is used to give an object of class sum a name; sum::getlabel returns this label (for printing). Sum::add updates the total. Notice that the procedures themselves assume they can access the strings directly. Sum::makelabel relies on the string bundler to pass it a pointer to the string, and sum::getlabel assumes that the string bundler will bundle its return value on the server side and unbundle it on the client. In this way, C++ semantics are preserved.

We also added two new type specifiers, *out* and *inout*, to the C++ formal parameter syntax; an *in* specifier is unnecessary because C++ already has a *const* specifier with the needed semantics. These specifiers are necessary for efficiently passing C++ pointers. If we wanted to pass only a pointer to a

---

```

class sum {
    int total;
    int num_entries;
    char* label;
public:
    void makelabel(const char*);
    char* getlabel();
    int add(int, int*);
};

// Newlabel is a const parameter, so it is only transferred in.
// The stringbundler is used to bundle the string newlabel points
// at. There is no return value.
void
sum::makelabel(const char* newlabel @ stringbundler())
{
    label = new char[strlen(newlabel)+1];
    (void) strcpy(label, newlabel);
}

// Getlabel returns the label that was stored by newlabel.
// Stringbundler is used once again to transfer the string.
// The procedure is written as if the caller and the sum
// class were in the same process.
char*
sum::getlabel() @ stringbundler()
{
    return label;
}

// Avg is an "out" parameter. The bundler provided by the compiler
// will be used.
// The return value is assumed to be an out parameter.
int
sum::add(int new_value, out int* avg)
{
    total += new_value;
    *avg = total / ++num_entries;

    return total;
}

```

---

Figure 2: C++ Procedure Declarations with Bundlers

remote procedure there would be no problem; however, sometimes we want to pass the data to which the pointer refers. The specifiers allow the parameter bundler to pass parameters in one direction rather than both, which is what C++ does normally (actually, C++ semantics pass pointers by reference, while with RPC, we can only pass parameters by value-result). Figure 2 shows an example of the use of these new specifiers. The `sum::add` procedure makes use of the `out` specifier. The `avg` parameter will only be passed from the server back to the client.

We consider the user bundler mechanism to be a limitation of the current implementation. One interesting alternative is to provide a *pointer fault* mechanism, whereby a reference to a pointer which is remote would cause the remote object or data structure to be copied only at that time. This is similar to the ideas being researched in distributed virtual memory [12, 13]. We plan to investigate this alternative in the future.

## 2.2. Programming Conventions for using RPC

We require certain programming conventions to make this work. First, all of the operations for a given object must be compiled together. This requirement simplifies the generation of the stub code that runs in the CLAM server. As a matter of style, we believe this requirement also to be a good idea. Second, only value parameters or explicit pointers are allowed to be passed to remote procedures; C++ reference parameters are not allowed. It is possible to allow remote reference parameters, but the programmer would have to supply bundlers for these parameters (as is done for explicit pointer variables).

The most important change is that all objects that belong to a dynamically loaded class must be declared dynamically. A new instance of an object is created by using the constructor for that class, so all dynamically loaded classes must contain constructor procedures. Static declared instances of objects for dynamically loaded class types are not permitted. This is because all object instances for a dynamically loaded class are stored in the server with the class. When a new instance of an object is created, a *handle* for that object instance is created. This handle is essentially a capability. The handle contains 4 parts: class identifier, protection key, version number, and pointer to the address (within the server) of the instance data structure. The class identifier specifies the class to which the object belongs. The protection key is a random number used to protect against simple access errors. The key can be considered part of the

class identifier. The version number is described in Section 3.3.

The remote procedure call mechanism for objects is well suited to the C++ language. The additions to the syntax are straight-forward, and the new types are useful in C++ in general, not just for the RPC mechanism. The programmer can access remote classes without adopting a totally different style of programming or using a different programming language.

### 3. Dynamic Loading and Binding

CLAM provides dynamic loading and binding of C++ classes. The dynamic loading and binding facility supports the goals of extensibility, sharing, and performance tuning. In addition to the basic facilities, the CLAM design includes features for protection and revisions and versions.

#### 3.1. System Structure

The CLAM server uses dynamically loaded classes to define its basic client interface. There are a few *built-in* classes, which are a static part of the server and are linked at server compile time, but all other functions are in dynamically loaded classes. A dynamically loaded class has access to all previously loaded classes and to the built-in library routines contained in the server. The *ControlClass*, which manages dynamic loading of new classes, is an example of a built-in class. The *ControlClass* also provides other client functions, such as querying the contents of the server. Other built-in classes define the lowest level display and input manipulation routines for the CLAM interface. All clients share these built-in classes, but each client is free to customize its specific environment by dynamically loading higher level classes. The actual loading and binding mechanism of the *ControlClass* is described in Section 3.3.

The built-in classes do not have to be identical between CLAM implementations. Except for the *ControlClass*, the client will rarely access the built-in classes directly. Most clients will build their applications on the basic CLAM library that is dynamically loaded when the server is started. This provides a uniform interface for clients while allowing for differences in the server implementation. For example, a workstation may have sophisticated display hardware, so it may be able to directly perform functions that another workstation may have to do in software. The goal is to keep actual built-in server code as small as possible, to reduce the likelihood of bugs and future changes. This is similar to the philosophy behind building a small operating system kernel.

### 3.2. Versions and Revisions

The user of a class may find that the class needs to be modified sometime after it has been loaded. The modification may be compatible with the current object data structure implementation or may be an incompatible change. A compatible change to a class is one that will still work correctly on previously created objects of that class. This type of change is called a *revision*. An incompatible change is called a *version*. These types of changes are similar to the facilities found in some database systems [14, 15]. The goal in CLAM is to make both revisions and versions transparent to the users of classes.

The version feature allows incompatible changes to be made in a class without affecting currently running clients. For example, consider the implementation of a queue or bounded-buffer. We may initially implement the queue as a linked list, but later decide to use an array implementation. Both class implementations have the same name and set of functions and can exist together. To keep the server and its internal data structures from growing too large, old versions are discarded when there are no remaining objects of that class and version.

Revisions are a simple change to the implementation of a class. All subsequent requests to that class will access the revision of the code. The old code is discarded after all requests that are currently in progress complete. To be compatible with the previous implementation of the class, revisions must be both data structure compatible and must have the same set of public functions.

### 3.3. Loading and Binding Mechanism

The mechanism that performs the dynamic loading is built into the CLAM server and is part of a special *ControlClass*. Through the use of this special class, a client can request that the server load a class. When the server receives a load class request, it locates the precompiled C++ class and then links and loads the code into the server, and returns to the client a handle for the newly loaded class. A client can only access classes in the server for which it has a handle. The dynamic loading of a class is a relatively expensive operation compared to calling a remote procedure, but it is done infrequently relative to subsequent accesses to the class.

If clients want to share dynamically loaded classes, then the server needs only to load the code for the class once. The code for a class is loaded the first time any client requests the class. Any subsequent

request from that client, or any other client, for loading that same class will be recognized by the server as unnecessary work since the class is already present. With this mechanism, sharing between clients is handled completely in the server. No extra communication between clients is necessary for them to share common classes.

An exception to the “no handle – no access” rule is needed for initially connecting a client to the server. To establish connections, the client calls the *OpenClam()* function, which is a library function linked in with each client. This function makes the initial connection to the CLAM server and requests a handle for an object of the *ControlClass*. The *OpenClam* function stores this handle in the private data area of the *ControlClass* stub. At this point the client is able to call functions in the *ControlClass* using this handle without any other special treatment.

All dynamic loading requests are made with the *ControlClass* handle using the *LoadClass()* function. Figure 3 shows the four forms of the *LoadClass()* function. The *LoadClass()* function allows a client to specify whether the class is a regular load request, a revision or a version. It also allows a client to specify the file from which to load the class. The server keeps track of the most recent version loaded. When a request for loading a new version is made (the fourth form in Figure 3), that version becomes the latest version. When a revision is loaded, by using the third form of *LoadClass()* from Figure 3, it replaces only the latest version of the specified class. Other versions remain unchanged. If the client specifies neither

---

```
// Load class "sum" from a standard place if the class is not
// already loaded, otherwise use the latest version.
void* sumhandle = Control->LoadClass("sum");

// Load class "sum" from the specified file if the class is not
// already loaded, otherwise use the latest version.
void* sumhandle = Control->LoadClass("sum", "file");

// Load a new revision of the class "sum" from the specified file,
// replacing the latest version of "sum".
void* sumhandle = Control->LoadClass("sum", "file", REVISION);

// Load a new version of the class "sum" from the specified file.
// This version becomes the latest version.
void* sumhandle = Control->LoadClass("sum", "file", VERSION);
```

---

Figure 3: Methods for loading a new class

revision nor version (first and second forms in Figure 3), the server will either load the new class if it is not yet loaded, or use the latest version of the class. The server maintains the different versions through the use of version numbers in the object handles. Clients that are using older versions continue to access their version, unaffected by new versions. Revisions and versions are transparent to clients.

There are two ways a client can request to load a class. The first is an implicit load request. This is automatically built into the client code for the constructors. When the constructor of a class is called, a check is made to see if the client already has a handle for that class. If the client does not have a handle for the class, a call is made to the *LoadClass()* function requesting the latest version. After the handle is obtained for the class the constructor, then the constructor function is actually called. The second way to load a class is through an explicit call to the *LoadClass()* function. The explicit call is useful for loading revisions or versions.

Handles are used for identifying both classes and individual objects. Class handles are stored in a private data area associated with each class stub in the client. The stub only requests a new class handle if it does not already have one. This has the side effect that a client accesses only one version of a class. Even if the client performs an explicit *LoadClass* a second time, the stub will not obtain the new handle and will continue to use the old version of the class. Object handles are stored on the heap in the client. They appear to the client code as opaque, structureless, data objects. Object handles are automatically created by object constructors and deallocated by destructors.

### 3.4. Protection

Three layers of protection are needed in a server based, user interface manager. These areas are authorization to access the system, restriction of privilege access once authorization is permitted, and error detection and robustness in the server. These three layers correspond to respectively lower levels of structure in the CLAM server.

The server must be protected from unauthorized access. Our current approach to this problem is to use the host based authorization provided by TCP/IP network connections, similar to the approach used in X [7]. There are two problems with this approach. First, the authorization is done on a per host basis and does not provide a fine enough level of granularity of protection if multiuser systems are used. Any user on

the authorized host can gain access to the server, where typically we wish to limit access to a select set of users. The second problem is that we rely on the network to provide host authentication. There is no guarantee that our communication media is secure and the current network protocols do not provide reliable authentication. Host authentication is a temporary mechanism that we are using in our prototype server. We are planning to base connection and authentication on a secure connection server, similar to [16].

After a process has permission to access the server, we may want to limit the classes, functions, and objects that it may access. For example, we may want to prevent the user from loading new classes or modify existing ones. We will also provide a *private* attribute for a class. A private class is one that can be accessed only by the creator of the class or by a user who receives a copy of a handle for (an object of) the class from the creator (much like capability). More work will be done in this area once CLAM starts using user authentication with the connection server.

A third area of protection is coping with execution errors. Once a process has the capability to modify or add classes to the server, we want to protect against that code violating the integrity of the server. While we cannot (currently) protect the internals of the server from damage by bad memory references, we can prevent some logic errors from crashing the server and disrupting service to other clients. Since we can not determine, a priori, that a given set of functions are error free, we must detect errors at run time. We are able to handle such errors as arithmetic errors (e.g., overflow and divide by zero), non-destructive memory faults, and system protection violations. Whenever such an error is detected, the server regains control and unloads the faulty class.

### 3.5. Debugging and Performance Tuning

A class can be statically linked with the client or dynamically loaded in the server. Code in the server is much more difficult to debug than code in the client. Testing and debugging of new classes can be done by first linking the class with the client and using the debugging tools provided by the operating system for regular C++ code. Once the routines are debugged they can be removed from the client and dynamically loaded into the server. Using the protection mechanism which unloads faulty classes, we can achieve a primitive debugging facility by informing the appropriate client when the class they were using

failed. We are currently investigating other types of debugging facilities that can be provided by the server.

The combination of a server model and dynamic loading permits us to tune system performance by carefully dividing the application into client and server parts so as to optimize communication and computation loads. Using the mechanisms for dynamic loading and knowledge about CLAM's environment, users can dynamically tune the performance of the system. There are two interesting cases to consider when tuning the performance. The first case deals with communications bandwidth. It is often desirable to decrease the amount of communication between the client and the server. Communication costs between the client and the server require interprocess communication overhead, while calls within the server are close to the cost of a procedure call. When the client and server are on different machines, this cost difference is increased. The second case is to consider the use of processing power. It is desirable to put the heavy computation where it can best be handled. For example, if the client was on a Cray-2 computer and the server was on a small workstation, then we would put as much computation as possible on the client. If the client was on an overloaded timesharing system and the server was on a powerful workstation, we might do the opposite.

CLAM allows the user to experiment with these options without changing programming semantics. The programmer has already divided the computation into well defined classes (using the object oriented structure of C++), so performance tuning is only a matter of where to place the classes.

#### **4. Handling Asynchronous Input**

The layering of output abstractions is well understood. For example, we might display three dimensional (3-D) objects by constructing them from 2-D objects, and constructing these objects from raster operations. Each request to display a high level object propagates down through the layers of abstraction.

Layering of input abstractions is more difficult. A common method for input in a layered system is to have an input request generated at the top level and propagating it down through the lower layers. This is essentially polling and requires that the input requests match the frequency of the actual input events, so as to provide interactive response to the input events. Input events include mouse and keyboard operations.

We would like input events to propagate upwards through the layers of abstraction, as the events occur. Each event causes changes in state to occur in layers up to a specified level. These changes are asynchronous, their timing depending only on the timing of the input events. Above the specified level, input requests are made in the same way as are output requests – synchronously. Each application or input abstraction determines the boundary defined by the level up to which asynchronous input propagates.

Two parts are needed to build the input system described above. The first part is the mechanism to call upwards through the layers of abstraction. CLAM uses a distributed upcall mechanism for propagating asynchronous input to higher level abstractions. Upcalls [9] allow procedure calls to be used when a lower level class wishes to call a higher level class. The distributed upcall mechanism can propagate these requests from the server back to the client.

The second part is a facility that provides multiple threads of execution in the server. This facility is used to support asynchronous input events. Threads of execution must also be able to span process boundaries.

#### 4.1. Distributed Upcalls

The concept of distributed upcalls is simple to understand. If an object of a higher level class wishes to be called asynchronously when some lower level event occurs, it *registers* its intent with an object of the lower level class. Registration is a matter of passing the address of a procedure in the higher level class to the registration procedure of an object for the lower level class. The low level object stores this address and a pointer to the higher level object in its own state. At a later time, when the appropriate event occurs, the lower level object will call the registered procedure to pass on this information to the higher level object. The information is passed upwards by a simple procedure call. The goal is to make upcalls work, without special effort by the programmer, even when the upcall requires a remote procedure call back to the client process.

Light-weight processes, called *tasks*[9], are used to provide the flow of control between objects. A new task is started whenever an asynchronous input event occurs and completes when the procedure handling the input terminates. Scheduling is non-preemptive for both simplicity and to assure that events are handled in the order in which they occur. Tasks may also block themselves. Upcalls and downcalls may

cross process boundaries, so the flow of control must also be able to cross the boundary between client and server. When an upcall crosses a process boundary, the task in the server is blocked and a corresponding task is created in the client. Control is given to the client task, which eventually returns a result to the object in the server. At this point, the task in the server is unblocked and continues. The use of tasks allows asynchronous events to propagate through as many layers as have registered procedures. CLAM places no limits on the level.

There are several problems in using distributed upcalls. The first problem is what an object should do if no higher level object has registered to receive the event. The answer depends on the semantics of the object. If no receivers are registered, the event could be queued and then sent on later when an object registered to receive it, the event could be thrown away, or the object could perform some arbitrary action depending on its state.

The second problem with implementing upcalls in an object-oriented environment is type-checking the parameters to a registered procedure. The object itself may not be able to have its type checked because its class might not exist when the lower level class was being compiled. We cannot rely on the compiler to guarantee that the types in an upcall are correct, so we need some kind of runtime support to provide this type checking. We do not have this support at this time, so we depend on the programmer to specify correct types.

The third problem with upcalls in a distributed environment is that pointers to procedures cannot be passed between processes. Our modified C++ compiler provides a special bundler for bundling procedure pointers. When a pointer to a procedure is unbundled in the server, this pointer is saved away in the state of a special distributed (remote) upcall (RUC) class object, and a pointer to a procedure within the RUC class is actually registered. When the procedure pointer is used, a procedure in the RUC class is called and it, in turn, makes a remote procedure call back to the client, receives return values, and returns them to the caller. While this is complex at the implementation level, the programmer is not required to use any special syntax to make the distributed upcall work. The compiler and runtime support automatically make distributed upcalls as easy to use as upcalls in a single process environment.

## 4.2. An Example

This section presents an example of the use of upcalls. Assume that there are two classes, *window* and *screen*, shown in Figure 4, and two additional application defined classes, *user1* and *user2*. *Screen* is a low level class that handles updates to the display screen. The *window* class provides a window abstraction layered over the *screen* abstraction. *User1* is a class linked into a client process and accesses the *window* class using a remote upcall. *User2* has been dynamically loaded into the server.

When the server begins execution, it creates an instance, *S*, of the *screen* class and an instance, *BaseW*, of the *window* class. While creating *BaseW*, the *window* class registers the *window::mouse* procedure with *S* (by calling *S.postinput*) to handle all mouse button events. *S.postinput* saves the pointer to

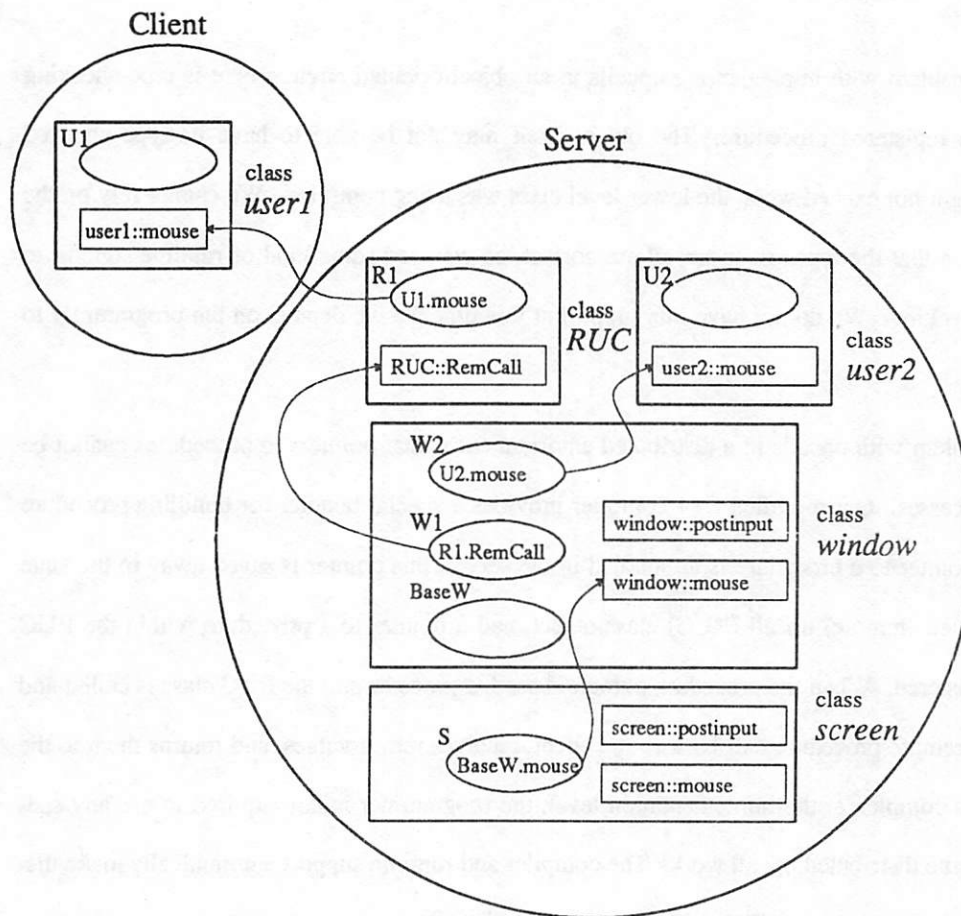


Figure 4: Registering Distributed Upcalls

BaseW and window::mouse in S's state. Later, an instance, U2, of the user2 class is created. It creates an instance, W2, of the window class and registers its *user2::mouse* procedure to receive mouse events by calling *W2.postinput*. Let us assume that creating W2 notifies BaseW of the new window, so it can pass events to objects that have registered themselves with W2. An instance, U1, of the client class user1 is also created. U1 creates a window, W1, and registers its *user1::mouse* procedure to receive mouse events. Notice that the parameter bundler will automatically translate the procedure pointer into a pointer to the RUC class. For each translation, an object instance is created in the RUC class.

At this point, the state of the system is ready to handle mouse events. If a mouse button is pressed, the *screen::mouse* procedure sees the event and, using the initial registration, makes an upcall to the BaseW.mouse procedure. This procedure determines if the mouse was inside any other windows and, if so, makes upcalls to them as well. If the mouse was in the region covered by W1, BaseW then makes an upcall to U1.mouse. This causes the RemCall procedure to make a remote procedure call to the client process containing U1.

As this example shows, the combination of distributed upcalls and normal downward procedure calls combine to provide a straight-forward flow of control between objects in an object-orient system. Down calls are not needed to get information from lower level abstractions.

## 5. Conclusion and Status

CLAM is based on the idea that complex systems are more easily built by providing a small collection of basic functions and a powerful means of composing these functions. The facilities in such a system are easily extended. Programmers write their extensions in the same language as they use for their applications. Programmers also have the flexibility to move these extensions between the client and the server.

A major part of the CLAM design is to allow users to abstract input as easily as output. This facility seems to provide a useful and powerful organization, but more experience is needed to test this design.

Protection in a single address space is a difficult problem. Some relief is offered by using compiler generated checks, but this is not bullet proof. This is an area for future study.

The current CLAM implementation is on a MicroVax-II workstation running 4.3BSD/NFS UNIX [1, 17]. The initial version of CLAM is now being tested. The C++ compiler has been modified to produce

the necessary code for remote procedure calls and the language has been extended to allow the user to specify bundler routines. The basic display routines and dynamic loading facility are running. One of the first large applications will be an X window library interface. This will allow us to use the collection of existing X-based applications until we can build more of these facilities directly into CLAM.

## REFERENCES

- [1] W. N. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher, *4.2BSD System Manual*, University of California, Berkeley (July 1983).
- [2] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass. (1983).
- [3] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass. (1986).
- [4] A. Black, N. Hutchinson, E. Jul, and H. Levy, "Object Structure in the Emerald System," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, Portland, OR, (October 1986).
- [5] D. B. Anderson, "Experience with Flamingo: A Distributed, Object-Oriented User Interface System," *Proc. of the Object-Oriented Programming Systems, Languages and Applications Conf.*, pp. 177-185 Portland, OR, (September 1986).
- [6] Sun Microsystems, Inc., *NeWS Preliminary Technical Overview*. October 1986.
- [7] J. Gettys, R. Newman, and T. Della Fera, *Xlib - C Language X Interface*, MIT Project Athena (November 1985).
- [8] Adobe Systems, Inc., *PostScript® Language Reference Manual*, Addison-Wesley, Reading, Mass. (1985).
- [9] D. Clark, "The Structuring of Systems Using Upcalls," *Proceedings of the 10th Symposium on Operating Systems Principles*, pp. 171-180 Orcas Island, WA, (October 1985).
- [10] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems* 2(1) pp. 39-59 (February 1984).
- [11] Sun Microsystems, Inc., *External Data Representation Protocol Specification*.
- [12] N. Carriero and D. Gelernter, "The S/Net's Linda Kernel," *ACM Trans. on Computer Systems* 4(2) pp. 110-129 (May 1986).
- [13] Kai Li, "Shared Virtual Memory on Loosely Coupled Multiprocessors," Technical Report YALEU/DCS/RR-492, Ph.D. Dissertation, Yale University (September 1986).
- [14] R. Katz and T. Lehman, "Storage Structures for Versions and Alternative," *IEEE Trans. on Software Engineering* 10(2)(March 1984).
- [15] A. H. Skarra and S. B. Zdonik, "The Management of Changing Types in an Object-Oriented Database," *Proc. of the Object-Oriented Programming Systems, Languages and Applications Conf.*, pp. 483-495 Portland, OR, (September 1986).
- [16] D. Draheim, B. P. Miller, and S. Snyder, "A Reliable and Secure UNIX Connection Service," *Proceedings of the Sixth Symposium on Reliability in Distributed Software and Database Systems*, Williamsburg, VA, (March 1987).
- [17] R. Sandberg, "The Design and Implementation of the Sun Network File System," *Usenix Association Summer Conference Proceedings*, pp. 119-130 Portland, OR, (June 1985).

1	Introduction
2	Background
3	Related Work
4	System Architecture
5	Implementation
6	Performance Evaluation
7	Conclusion
8	Acknowledgments
9	References
10	Appendix A
11	Appendix B
12	Appendix C
13	Appendix D
14	Appendix E
15	Appendix F
16	Appendix G
17	Appendix H
18	Appendix I
19	Appendix J
20	Appendix K
21	Appendix L
22	Appendix M
23	Appendix N
24	Appendix O
25	Appendix P
26	Appendix Q
27	Appendix R
28	Appendix S
29	Appendix T
30	Appendix U
31	Appendix V
32	Appendix W
33	Appendix X
34	Appendix Y
35	Appendix Z

# Experience In Using C++ For Software System Development

William E. Hopkins

AT&T

11900 N. Pecos St., Denver, CO 80234

## ABSTRACT

The prevalent software development approach in industry is design using functional decomposition and implementation using a standard programming language that supports only functional abstractions (e.g. Pascal, Fortran, and C). This approach has many well known deficiencies. In an attempt to lessen or eliminate many of those deficiencies, object-oriented design coupled with the C++ programming language were used to develop a new release of a software configuration management system.

Many benefits resulted from using object-oriented design and C++: a more comprehensible design and implementation, better modularity, easier to modify software, increased reliability, and more software reuse. All these benefits contribute to higher productivity and better quality.

Regarding C++ itself, the language, implementation, and documentation, although not fully mature when the effort began, are now developed to production quality, and thus, are adequate for large scale commercial software development.

# Experience In Using C++ For Software System Development

William E. Hopkins

AT&T

11900 N. Pecos St., Denver, CO 80234

## 1. Introduction

As a result of the author's experience of developing software at AT&T for four years, it became apparent that the C programming language <sup>[1]</sup> and the functional decomposition design approach have some inherent shortcomings. Function interfaces are not verified<sup>1</sup>, making interface-mismatch errors difficult to detect and integration with other software tedious. Manipulations of objects<sup>2</sup> are not limited to operations separately implemented and bound to the object; it was more frequent that objects were manipulated directly by the code which uses the objects, accessing their internal representations directly. The result was that dependencies on the most minor and local design decisions were scattered throughout a system, which made the modification and reuse of software difficult. Even when objects were designed using established principles (e.g., information hiding <sup>[4]</sup>), the language made them tedious to implement since it provided no support for describing them.

In the environment described above, much of the time was spent dealing with the problems inherent in the tools and the approach rather than with those of the problem domain addressed by the system being developed. As a result, there was interest in finding new tools and approaches which would let one concentrate more on the system.

A new variant of C <sup>[5]</sup> was developed that offered some potential for addressing some of the problems. It became C++ <sup>[6]</sup>, and was being used in various exploratory projects. A C++ interest group was formed within AT&T during the summer of 1984, which resulted in the author making contacts with those involved with C++ and obtaining a copy of the C++ translation system.

To evaluate C++, and what has come to be known as object-oriented design, they were used in the development of a new version of MESA<sup>3</sup>, release 2.0. This paper is a short description of the lessons learned from that experience.

Below is a brief discussion of the principles of object-oriented design, and a list of the main features of C++ along with their advantages over C. Both are derived from the MESA 2.0 development experience, and include examples from MESA 2.0 C++ code. Following are a summary of the benefits of using C++ in software projects, a discussion of the problems encountered as a result of using C++, and conclusions.

## 2. Object-Oriented Design

Object-oriented design is an approach to software design in which the designer focuses on the objects of the problem domain and the kinds of operations that can be performed on them, rather than focusing on processes and functions <sup>[7]</sup>. The result of decomposition, then, is a collection of object definitions rather than a set of steps in the process.

Two concepts upon which object-oriented design are based are information hiding <sup>[4]</sup> and abstract data types <sup>[8]</sup>. Object-oriented design extends the abstract data type approach by allowing new object

1. Separate analyzers are available (e.g., `lint` <sup>[2]</sup>) for verifying function interfaces, but their use requires separate invocation and, in the case of `lint`, maintenance of separate libraries of interface definitions. In practice, such analyzers are often not used due to the extra overhead and configuration problems the libraries present.
2. The definition used here for *object* implies only an entity which has state and has operations defined on it, even if those operations are implicitly defined in code in which it is embedded. Another common use of *object* is to mean the combined description of state and operation <sup>[3]</sup>.
3. MESA (Management Environment for Software Administration) is a software configuration management system developed for internal use at AT&T.

types to be derived from other types. The result is that the relationships of specialization and generalization between object types are described <sup>[9]</sup>. Describing these relationships in design makes reusability of common object descriptions possible, and given a language that supports this, likewise for source code.

The main advantage afforded by object-oriented design is that the system structure reflects that of the problem domain. This makes the system easier to develop and understand, and provides a useful method for decomposing development into separate pieces.

Since the interfaces to objects reflect the abstract concepts being modelled rather than the implementation, changes to lower level details are more easily localized to a single object type, making the system easier to modify<sup>4</sup>.

In MESA 2.0, two basic object types were `WorkFile` and `WorkArea`, reflecting the "real world" MESA objects of work areas and work files. The kinds of operations defined on them are the kinds of things one does to work files and work areas (e.g., reserve for editing, record new version). All information on how these operations are carried out is contained within their definitions, so any change to fix a problem with or to improve the implementation of those object types will be localized to their definitions rather than being scattered throughout the system wherever they are used.

### 3. C++ Features

Below are brief descriptions of the main C++ features (those that distinguish it from C), what benefit can be derived from them, and an example of their use in MESA 2.0.

#### 3.1 Classes

The construct that supports object-oriented programming in C++ is the `class`. It is an extension of the concept of a `struct`<sup>5</sup> (in C++, a `struct` becomes a special case of a class which has no private members). A class can have not only data members, but also *member functions*. It allows the programmer to define new data types complete with operations on those data types.

Below is a discussion of four of the main features of classes: provision for a public interface and private implementation, constructors and destructors for class initialization and cleanup, type inheritance, and dynamic typing.

##### 3.1.1 Public interface/private implementation

The members of a class, both data and functions, can be made either private or public<sup>6</sup>. Public members define the interface of an object of that class to client code. Access to private members is restricted to the class member functions and to those functions and classes explicitly granted access via the `friend` declaration specifier. This is the mechanism that enforces information hiding and provides the means for defining abstract interfaces.

An example of how public and private class members are denoted in C++ is given in figure 1. In the example, the class, `PathList`, is declared; the class members listed after the opening curly brace but before "`public:`" are all private (`length`, `rep`, and `ptr`), and the class members following "`public:`" are public (`addpath()`, `next()`, `search()`, `reset()` and the constructor and

4. Parnas explains this approach in terms of hiding design decisions behind module specifications. For a lucid explanation of how this design approach results in systems being more modifiable (in addition to when to use it), see <sup>[10]</sup>.

5. A `struct` in C is the analog of a `record` in Pascal: it allows the definition of an aggregation of heterogeneous data elements which are referenced by the (field) name. In contrast, an array is an aggregation of homogeneous data elements referenced by position.

6. A new access classification, denoted by the keyword, `protected`, has since been added to C++ (for more details, see <sup>[11]</sup>). Access to `protected` class members is limited to the member functions of the class and of all derived classes (see section 3.1.3), and to friends. This feature was included in the AT&T C++ Translation System Release 1.1.

destructor [see section 3.1.2]). The function, `operator <<()` [see section 3.3], is a friend of `PathList`, meaning that it can access the private members of a `PathList` object. Please note that in addition to the C comment syntax of `/* <comment> */`, C++ has `// <comment not containing end-of-line> <end-of-line>`.

```
class PathList
{
    friend ostream& operator<<(ostream&, PathList&);

    int    length;
    char*  rep;
    char*  ptr;
public:
    PathList& addpath(FilePath&);           // add path to end of list
    int    next(FilePath&);                // return next path and
                                           // advance ptr
    void    reset();                       // set pointer to beginning
    int    search(FilePath& srch_obj, AccessType, FilePath& object_path);
                                           // search for item in Path

    PathList(char* pl = 0);                // initialize from path:path:path
    ~PathList();
};
```

Figure 1. Class declaration with public and private members

In MESA, most, if not all, of the design consisted in first identifying the object types (at the various levels of abstraction) and their operations, which were expressed by declaring a class with public member functions corresponding to the operations. The implementation of the object type followed, declaring the necessary private data items and private member functions, and defining the public member functions. It was a simple matter to create skeleton implementations of classes which would, for example, print out the name of the member function to allow the early testing of code which used objects of those classes.

One major benefit of having public and private members is that the interface to an object can be in abstract terms only, removing implementation dependencies from all but the class definition. As a result, radical change can take place in the implementation of an object type without having the effects ripple throughout the system.

### 3.1.2 Constructors and destructors

C++ allows one to define two special member functions for classes: constructors and destructors. They provide automatic initialization and cleanup at the creation and destruction of an object. This relieves the user of the responsibility to invoke initialization and cleanup functions. In addition, constructors and destructors assure the implementor of the class that all objects of that class will be initialized to a known state, and that all necessary cleanup (such as memory deallocation) will be done.

In MESA 2.0, all classes have constructors defined, and most have destructors. There are some interesting cases worth noting. MESA 2.0 has a debugging facility for tracing member function execution: one can assign to a UNIX<sup>TM</sup> shell environment variable<sup>7</sup> the names of the classes to trace with

<sup>TM</sup> UNIX is a registered trademark of AT&T.

7. The shell is the UNIX command interpreter. Within the shell, one can define variables, assign them values, and access their values from within UNIX processes.

the result that those member functions with the trace facility will note to the standard UNIX error output stream when they are entered and left, and will print out any other specified trace messages. To provide this feature for a member function, the first line in the member function body must be a definition of a Trace object, as done below in figure 2:

```
Boolean
WorkFile::reserve_file()
{
    Trace trace(TR_WorkFile, "reserve_file");
    // ...
}
```

Figure 2. Definition of a Trace object

This one line results in the printing of an entry message and an exit message, accomplished by the Trace constructor, which is called at the point the object is defined, and by the Trace destructor, which is called when the function is exited (since the object goes out of scope). The Trace class is declared in figure 3. Class constructors and destructors are declared in the same manner as other member functions; they are distinguished by having the same name as that of the class, in this case Trace (destructor names are preceded by "~").

```
class Trace
{
    // ...
public:
    // ...

    Trace(TraceClass, char*, char* = 0);
    // check whether class is on, entry msg
    ~Trace(); // send out exit message if on
};
```

Figure 3. Declaration of Trace showing constructor and destructor

The implementation of the trace facility makes another use of a constructor in a statically defined object of type `TraceList`. Constructors for statically defined objects are called at the beginning of process execution, and the destructors are called immediately before the process is exited. For a `TraceList`, the constructor looks for the shell environment variable `TRACE`. If it exists and has a value, the constructor looks for the names of the known classes (entered by the programmer), and records which classes should be traced. The Trace constructor checks the `TraceList` (a simple array access indexed using a constant identifying the class) to see if the specified class is being traced. Processes do not need to reference explicitly a `TraceList` object, much less be responsible for initializing it. The `TraceList` object is automatically included whenever a `Trace` object is used.

### 3.1.3 Type inheritance

Type inheritance is the concept of one type inheriting the characteristics of some other type, usually for making a specialized version of the base type. In C++, this is accomplished by declaring a class as being derived from the another class<sup>8</sup> (see figure 4). The obvious advantage is that of reusability of

```

class SingLinkList
{
    Boolean          sortable;          // are the elements in sortable storage
    SingLink*        last;              // last->next points to head of list
    unsigned          member_count;
public:
    int              append(void*);      // add entry at tail of list

    // ...
};

class LineOfDescent: SingLinkList
{
    VersionNode*     base_version;
public:
    int              append(VersionNode* vnode)
                    { return SingLinkList::append(vnode); }

    // ...
};

class LodList: SingLinkList
{
    // ...
public:
    int              append(LineOfDescent* lod)
                    { return SingLinkList::append(lod); }

    // ...
};

```

Figure 4. Example of type inheritance

existing classes for more specialized applications. One avoids having to copy the base class into the new one (and so on up the type inheritance tree), and avoids the headaches that accompany having to maintain multiple copies of the same source. In addition, an improvement or fix to the base class will be automatically included in its derived classes.

One example of reuse is the single linked list class, `SingLinkList`. It was implemented such that it could take a pointer (a `void*`) to any kind of object, and was tested for correctness. To have lists of particular types (which eliminates casting and provides type checking), `SingLinkList` was made the base class for the special lists, e.g., `LodList` and `ReservationList`. This avoided having to rewrite and retest the list facility for each special list. Figure 4 contains some code fragments which illustrate how `SingLinkList` is reused in the declaration of two specialized lists, `LineOfDescent` and `LodList`.

### 3.1.4 Dynamic typing

Dynamic typing in C++ provides the ability to have an operation performed dependent on the type of the object as determined at runtime. This is the facility that makes C++ a truly object-oriented programming language<sup>[3]</sup> (in addition to data encapsulation with type inheritance).

8. The class from which another is derived is known as the *base* class of the derived class.

The language facility which makes dynamic typing possible is that of virtual member functions in conjunction with type inheritance. The situations in which this facility is most useful are those in which a version of some operation to perform on an object is selected according to the type of the object. When implemented using C, there is usually a `switch` on a field indicating the type, and a `case` for each type. Each case contains the version of the operation which is specific to the type selected by the case label. This kind of construct is typically found all over the system, so any new type necessitates modifying each occurrence, bringing the possibility of missing some instances<sup>9</sup> and also of inserting new errors.

Using C++, one can replace each of those `switch` statements by a common base class object reference and member function invocation, and by declaring that member function as being `virtual`. A class is derived from the common base class for each of the types of data objects which would have been denoted by the type field. In each of the derived classes, the base class virtual member function is redefined to perform the operation in a manner which is specific to that derived class. C++ virtual member functions perform the same function as the `switch` statement, the differences being that the compiler handles the addition of new cases, and the operation selection is more efficient (one memory indirection).

A major benefit of dynamic typing is that of code modifiability. New derived classes may be defined and added to the system without having to modify the client code since the objects of the new derived classes can be accepted wherever objects of their base class are referenced. All that must be done is to include the new derived class declarations in the source file and to recompile the file.

One other advantage is that of comprehensibility, since the semantics of having a collection of types which have commonality at a higher level of abstraction is made explicit via type inheritance. Comprehensibility is also enhanced by the code not being cluttered by big `switch` constructs.

Reliability is increased, since there is no possibility of forgetting to add a case to a `switch` statement for handling all the possible types of objects.

One example of dynamic typing in MESA 2.0 is the use of classes to represent files in the UNIX file system, shown in figure 5. Class `UnixFile` has member functions defined on it, including `create` and `remove`. The same implementation of those functions will work on most files, with the notable exception of directories. Thus, both `create` and `remove` are declared as being `virtual`, and a class is derived from `UnixFile`, called `Directory`, which implements the `create` and `remove` operations in a manner that is appropriate for directories. Code segments that create and remove `UnixFile` objects can now also properly handle `Directory` objects, without any modification.

### 3.2 Function argument type checking

C++ requires that the argument types in all function invocations match the types in the corresponding declaration. Thus, all function declarations must include specification of the argument types<sup>10</sup>.

The first and most obvious benefit of function argument type checking is that there will *never* be any errors due to type mismatching between function definitions and function invocations.

Since this checking is done at the time of compilation, it has the added benefit that a second analyzer such as `lint`<sup>11</sup> is not needed, and that by the elimination of the second analyzer, such checking occurs for each compilation, not just occasionally (or sometimes not at all).

9. If the type field is denoted by an enumerated type, the C++ translator will give a warning if most but not all the values of the enumerated type are represented by a case in the `switch` statement.

10. This feature of C++ was considered valuable enough to be included in the proposed American National Standards Institute (ANSI) standard for C<sup>[12]</sup>.

11. `Lint`<sup>[2]</sup> checks C programs for syntax errors, type violations, portability problems, and a variety of probable errors.

```

class UnixFile
{
    // ...
public:
    virtual Boolean    create();    // create file
    virtual Boolean    remove();    // remove file from file system
};

class Directory: public UnixFile
{
    // ...
public:
    Boolean    create();            // create directory
    Boolean    remove();            // remove directory from file system
    // ...
};

void dosomething(UnixFile& ufile)
{
    // ...
    ufile.create();
}

void some_function()
{
    UnixFile    ufile(fpath);
    dosomething(ufile);            // dosomething will call UnixFile::create

    Directory    dfile(dpath);
    dosomething(dfile);            // dosomething will call Directory::create
}

```

Figure 5. Example of dynamic typing

The experience in MESA 2.0 was that there was only one error due to an interface problem. That occurred for an invocation of a function which had two consecutive arguments of the same type, and the actual parameters were reversed. The declaration of the function is given in figure 6.

```

Boolean next_version(VersionDbase*,VersionEnv&,Boolean,Boolean,
                    VersionId&,VersionId&);

```

Figure 6. Example of a function declaration

The two Boolean arguments were reversed in the invocation. If concerned about that kind of error, one could use separate type declarations, via the class mechanism, to differentiate between two objects whose underlying representations are the same.

### 3.3 Operator overloading

Both operators and function names may be overloaded, i.e., more than one definition may be provided for a given operator or function name as long as they can be differentiated by the argument types. An added restriction to overloading operators is that at least one of the arguments must be of a

```

class VersionId
{
    VersionIdState state;
    short rel, lev, brn, seq;    // release, level, branch, sequence
public:
    VersionId& operator++();    // increment to next sequential vid
    Boolean operator==(VersionId&);
    VersionId& operator=(VersionId&);
    Boolean operator!()    // is the object not good
                        { return ! state.isgood(); }

    // ...

    VersionId(char* vid_str = 0);
    VersionId(int r, int l, int b = 0, int s = 0)
        { rel = r; lev = l; brn = b; seq = s; }
    VersionId(VersionId&);
};

```

Figure 7. Overloading of both operators and constructors

class object type<sup>12</sup>.

The two main benefits to overloading are that one can extend the meaning of common operators or functions to new types, and that the same function can be redefined to accept different sets of arguments where appropriate.

```

overload chmod;
overload creat;
overload link;
overload unlink;

extern int  chmod(char*,int);
extern int  chmod(FilePath&,int);
extern int  creat(char*,int);
extern int  creat(FilePath&,int);
extern int  link(char*,char*);
extern int  link(FilePath&,FilePath&);
extern int  unlink(char*);
extern int  unlink(FilePath&);

```

Figure 8. Declarations for overloading standard UNIX functions

For an example of the first usage, many of the classes in MESA 2.0 include definitions for the assignment operator and the equality operator. The second usage is demonstrated in many, if not all, MESA 2.0 class declarations since most constructors provide more than one interface. Shown in figure 7 is a class declaration showing both uses of overloading. In figure 8 is the declaration of functions

12. This restriction ensures that someone cannot change the meaning of an expression consisting of only predefined types (see section 6.2.3 of [13])

overloading standard UNIX functions (non-member functions must be explicitly declared as being overloaded by the `override` function specifier), and an example of the implementation of overloading an operator, `<<`, is given in figure 9 (an operator, for the purposes of declaring and defining, is denoted as a function with the name, "operator <op symbol>").

### 3.4 References

In C++, one can declare an identifier as being another name for some existing object; such an identifier is known as a reference. It is like a pointer in that it references an existing object, but it behaves like the name of the object for all purposes.

There are two main advantages of using references. The first advantage is from their use as formal parameters for a function into which the objects being passed are large or they are to be modified as part of the function output. One gets the efficiency and semantics of passing a pointer along with the syntactical convenience of treating it as an object.

The other advantage is from their use as the return value of a function. This allows functions to be on the left-hand side of an assignment.

```
ostream&
operator<<(
    ostream&    out, // output stream onto which to write the mesaline
    Mesaline&    mline // object from which to get mesaline info
)
{
    if ( !out ) return out;

    out << "MESA:01:@(#):" << *mline.mid << ":" << *mline.project << ":"
        << *mline.vid << ":" << mline.date << ":" << mline.machine << ":"
        << "1 1 1" << ":MESA>";
    return out;
}
```

Figure 9. Example of using references

Most of the examples included in this paper have object references being passed into functions. The example for operator overloading illustrates both passing in and passing out references. One case where passing out a reference is of absolute necessity is for overloaded operators which return one of the objects which is to be acted upon by another operator. This is clear in the example in figure 9, which shows the use of the Stream I/O system which comes with C++<sup>13</sup>. The `<<` operator is overloaded to mean "write the right operand to the output stream defined by the left operand." In the long statement which writes to `out`, each `<<` takes two operands, an `ostream` and some other object, writes to the output stream according to how `<<` is defined for that pair of operands, then returns a reference to the `ostream`<sup>14</sup>.

13. The Stream I/O system illustrates how C++ can be used to create new types for performing useful tasks. Stream I/O has the advantage over the standard C library `printf` and `scanf` functions of providing a type secure and uniform I/O approach for all objects since it is easily extendible to include user-defined types. For more details, consult [14].

14. The reason for the `<<` operator returning an `ostream` reference is that `out` must be the left operand to all of the `<<` operations. `ostream` objects contain variables, such as buffer pointers, which are adjusted by `<<` operations. Since `out` is likely to be used in other `<<` expressions, it must be the left operand to all of the `<<` operations, which wouldn't be the case if the `<<` operator returned an `ostream` object value.

### 3.5 Inline function expansion

For avoiding the overhead of function calls, C++ provides the means of having functions expanded inline. The use of abstract interfaces to objects usually results in a number of small functions which performs only trivial operations such as returning the value of a private data member. For those cases in which the function call overhead is significant, one can use inline function expansion since it reduces execution overhead to that of including the code in place of the call. The inline function expansion capability should be used judiciously since it can expand the program text by the size of the function text multiplied by the number of invocations minus the overhead that would have been incurred if actual function calls were used. Note, however, that for most conventional machine architectures, this implies that inline expansion will actually cause the program text to shrink for very small functions like those in figure 10.

```
class VersionEnv
{
    VersionId      vid;           // work file version id
    unsigned short release;       // release of the work area
    Boolean        old_rel;       // work area old release status
public:
    VersionId&      get_vid()      { return vid; }
    int             get_release() { return release; }
    Boolean         get_old_rel() { return old_rel; }

    VersionEnv(
        VersionId&    in_vid,
        int           in_release,
        Boolean        in_old_rel
    ): vid(in_vid)      { release = in_release;
                        old_rel = in_old_rel; }
};
```

Figure 10. Inline functions within a class

The inline function expansion facility removes one of the most common objections to language enforced data encapsulation, namely, function call overhead (execution performance seems to be of paramount importance in *all* software development projects<sup>15</sup>).

In addition to removing the overhead for accessing object data, inline function expansion eliminates the need for functional macros, with the added benefit that type checking will be performed. One other problem with macros that is eliminated by using inline functions is that of subtle bugs due to the influence of the context, e.g., precedence of operators in the context being higher than those in a poorly formed macro expression.

Examples of inline functions abound in MESA 2.0 since it is easier to include a simple member function inline which only returns a value, than to write an entire separate definition. The example in figure 10 is a simple class which is implemented entirely in the class declaration, and the example in figure 11 shows a simple use of inline functions for overloading standard UNIX functions (to declare a function as an inline, it must either be preceded by the function specifier, `inline`, as in figure 11, or be a member function with the definition given in the class declaration as in figure 10).

15. "Requirements expand until the program exceeds the available memory and cpu capacity."

```
// define common overloaded functions as inlines
inline int  chmod(FilePath& fpath,int mode)
    { return chmod(fpath.rep,mode); }
inline int  creat(FilePath& fpath,int mode)
    { return creat(fpath.rep,mode); }
inline int  link(FilePath& existing,FilePath& newfile)
    { return link(existing.rep,newfile.rep); }
inline int  unlink(FilePath& fpath)
    { return unlink(fpath.rep); }
```

Figure 11. Inline definition of overloaded functions in figure 8

### 3.6 Constants

To include constant values in program text, one prefers to use some kind of symbolic representation which is part of the language and is defined in one place so that its value can be changed without having to edit all the places where it is used. C++ introduced constants to support this<sup>16</sup>.

A major benefit is that constants are subject to type and syntax checking, unlike the preprocessor macros often used in C. One can also declare formal arguments as being constant<sup>17</sup>, which indicates to the function user that the value passed into the function will not be modified, and it prevents the function body from modifying it. In addition, one can define a function return type as being a constant, the effect of which is to require that the object receiving the return value be a constant. It can also be used to return a reference or pointer to a "read-only" object. This capability can be quite useful in defining interfaces so that both users and implementors will be aware of what to expect and will be prevented from violating it.

```
const int    MesaNameLen = 13;                // including null char

class MesaName
{
    char      rep[MesaNameLen];
public:
    int        operator==(MesaName&);          // equality
    Boolean    operator!=(MesaName&);          // inequality
    MesaName&  operator=(MesaName&);          // assignment

    MesaName(char* first_value = 0);
    MesaName(MesaName&);
};
```

Figure 12. Use of constant to define set value

In MESA 2.0, constants were used to provide symbolic references to constant values, but no use of them was made in the role of defining function interfaces<sup>18</sup>. In figure 12 is an example of the use of a

16. This feature also was adopted into the ANSI standard for C <sup>[12]</sup>.

17. Since arguments are by default passed by value, declaring arguments as constants is relevant for references and values passed by pointers.

18. The main reason for not using constants in function interfaces was that they were not fully understood when the project started (the C++ book <sup>[13]</sup> wasn't available yet), and that with a development team of one, there weren't too many problems in communicating expected interfaces.

```

char *strcpy(char*, const char*);
char *strncpy(char*, const char*, int);
char *strcat(char*, const char*);
char *strchr(const char*, char);
char *strrchr(const char*, char);
char *strtok(char*, const char*);

```

**Figure 13.** Use of constant in defining function interface

constant in a simple class declaration.

Although MESA 2.0 didn't use any constants as function arguments, there are some in the C++ include files, as given in figure 13, for the standard string function library.

### 3.7 Default function argument values

It is possible to provide default values for arguments not given in a function invocation. The default value is an expression specified in the function declaration which is evaluated at the time of the call.

The principal value of default function argument values is that of notational convenience. It is an alternative to having to overload the function for cases where not all arguments need be specified and to having all invocations give a commonly used value.

```

class HistoryFile: public UnixFile
{
    // ...
public:
    Boolean replace(
        VersionId&, // vid to specify the reservation
        char* = 0   // comment to associate with new version
    );
    // ...
};

```

**Figure 14.** Example of default value to avoid unnecessary overloading

An example of the first use is in the `HistoryFile` class for the `replace` member function. Not all invocations need give a comment, so the default is to set the character pointer to zero (see figure 14).

```

class VersionDbase
{
    // ...
public:
    void dump(ostream& = cerr);
    // ...
};

```

**Figure 15.** Use of default value to specify commonly used value

The second use is illustrated by the diagnostic `dump` member function of the `VersionDbase` class. The default output stream is `cerr`, but it can be overridden by any particular invocation (see figure 15).

### 3.8 More and better diagnostics

In addition to having the features outlined above provided by the language, the C++ translator gives more helpful diagnostics than the PCC2 C compiler <sup>[15]</sup>. Syntax checking, in general, is better than that of the C compiler, e.g., in many cases, it attempts to explain the nature of a syntax error beyond the message, "syntax error."

Warning messages are given for variables defined but not used and variables used but not initialized (only within the context of a single function; thus it doesn't provide the more global information that is generated by a dataflow analyzer <sup>[16]</sup>).

The translator also gives a warning when a switch on an enumerated type variable includes most but not all values as labels in case statements. Thus, the translator detects those switch statements which may need to be modified after adding a new value to an enumerated type.

Overall, the translator performs many more diagnostics than the C compiler, and most of what lint does, thus helping developers find potentially troublesome (and most likely subtle) problems in the code.

### 4. Summary of Benefits to Software Projects

Below is a summary of software system attributes improved by using C++ and object-oriented design, associated with those features that provide the improvement.

system attribute	feature
comprehensibility	<ul style="list-style-type: none"> <li>• system structure reflects that of the problem domain</li> <li>• constructors/destructors eliminate clutter due to initialization and cleanup function calls</li> <li>• type inheritance makes commonality explicit</li> <li>• dynamic typing removes code cluttering switch statements</li> <li>• overloading allows common operators to be extended to new types</li> <li>• overloading to extend functions to accept different argument sets</li> <li>• default arguments reduce code clutter from common argument values</li> </ul>
modularity	<ul style="list-style-type: none"> <li>• system consists of a collection of relatively independent objects</li> </ul>
modifiability	<ul style="list-style-type: none"> <li>• changes to object implementation localized (design decisions hidden within module)</li> <li>• public interface/private implementation enforces isolation of object internals</li> <li>• type inheritance propagates base type changes to derived types</li> <li>• dynamic typing allows addition of new derived types without changing client code</li> </ul>
reliability	<ul style="list-style-type: none"> <li>• constructors/destructors guarantee proper initialization and cleanup of objects</li> <li>• function argument type checking ensures that the proper set of arguments according to number and type are sent</li> <li>• inline function expansion makes error-prone macros unnecessary</li> <li>• constants allows type and syntax checking as opposed to using macros for the same purpose</li> <li>• constant function parameter types and return types provide more enforcement of the interface</li> <li>• more and better diagnostics pinpoint many potential sources of problems at compile time</li> </ul>
reusability	<ul style="list-style-type: none"> <li>• independent, self-contained objects are more easily reused</li> <li>• type inheritance increases the reuse of software by allowing new classes to be derived from existing ones</li> </ul>

## 5. Problems

The use of C++ provided many benefits, but it also was the source of some problems. These problems were mainly due to the immaturity of the language support at the time and have been resolved since. They are included here to give an example of the kinds of problems one encounters when working with new technology. Below is a summary of those problems.

### 5.1 Local support

The first and most significant problem is that there was no local support, meaning that the author had to provide it. Thus, he had to integrate in new versions of the C++ translation system, port it to the different machines used<sup>19</sup>, and report bugs to those who provided "remote" support<sup>20</sup>.

19. The portability of the C++ translation system was quite useful: during the course of the development effort, it was ported, along with the system being developed, to a DEC VAX™ 11/780, an AT&T 3B20, an Amdahl 5860, and finally to an AT&T 3B2, which were all running UNIX System V Release 2. Given a reasonably good means of transferring data to it, the C++ translation system can be ported to a different machine in a day.

™ DEC and VAX are trademarks of Digital Equipment Corporation

20. This was before the C++ Translation System was supported as an external product of AT&T.

Local support is no longer a problem for various reasons: most of the known bugs have been removed; it is now even easier to port; someone is available to provide any necessary local support (e.g., distribution and installation); and the product is fully supported by AT&T.

## 5.2 Incomplete language information

Another problem was that the project started without a complete source of information on the language. There was a collection of articles and memos written which helped explain some aspects of the language, but they were either incomplete or out of date. In addition, there was no one available locally who could answer questions.

The availability of the C++ book <sup>[13]</sup> and of C++ training courses now satisfies the need for a source of information. The lack of local expertise in C++ is being solved by the increase in the number of people gaining experience and becoming "experts" for new programmers.

## 5.3 Preprocessor implementation

The current C++ translation system converts C++ code into C code, which is then compiled using the C compiler. This results in extra overhead when compiling C++ code but has the advantages that it finds errors much quicker than the C compiler, it is highly portable and retargetable, and can be used easily for cross-compilation. For ongoing development, the extra time spent in compilation may be unacceptable to many programmers. This problem will be solved when native C++ compilers become available, while the translator will still be available for cross-compilation and for those machines for which no native C++ compiler yet exists.

## 5.4 Lack of C++ tools

Many of the standard UNIX tools that help with using C do not work on C++, and their C++ equivalents are not available. Such things as `cxref` and `cflow`, the UNIX C program cross-reference and flow graph generators, are not yet available for C++. The UNIX symbolic debugger, `sdb`, works on C++ code in a limited way, but it requires a knowledge of how C++ is translated into C.

This will take longer to resolve, but the increasing popularity of C++ will guarantee that something will be done. The `pi` debugger<sup>[17]</sup> developed at AT&T Bell Laboratories by Tom Cargill handles C++ quite nicely. In addition, there is a version of the C program `cscope`<sup>[18]</sup> for C++, called "`cscope++`."

## 5.5 Standard Libraries

To realize the full benefit in terms of reusability, there needs to be support for object-based access to common objects in the programming environment, namely UNIX. For example, in MESA 2.0, classes for things such as file paths and UNIX files had to be defined. With no libraries available which provide such classes, developers will all spend some time in defining and testing them.

There is some effort underway right now to identify what would be a useful set of standard classes and to make them part of the C++ environment.

## 6. Conclusions

The experience using C++ was most positive. Even though the author did not take full advantage of the language<sup>21</sup>, good design was facilitated and many common errors were avoided. The one facility that helped the most was the ability, via classes, to express directly, and have the compiler enforce, data abstractions.

There is a gap between those languages which are highly practical and efficient but provide little for expressing higher level abstractions and those languages which are rich in abstraction facilities but are

---

21. Making the switch from the procedure oriented paradigm to object oriented paradigm requires a significant change in the way one approaches software design, which comes only with time.

too inefficient to be practical. C++ bridges that gap, making all the benefits of higher level abstraction facilities available for use in production software development.

## 7. Acknowledgements

First, I would like to thank Jim Coplien, Jonathan Shopiro, and Bjarne Stroustrup; they provided the necessary support back in the early days of the language. I would also like to thank those who reviewed an internal version of this paper: Jim Coplien, Pete Kirsliis, Andy Klein, Jonathan Shopiro, Bjarne Stroustrup, and Tom Williams. They made many useful suggestions, most of which are manifest in the final draft. Lastly, I would like to acknowledge Larry Rosler and Alan Glasser, who both greatly influenced the content and style of this paper, making it more meaningful to those not necessarily immersed in the AT&T/UNIX environment.

## REFERENCES

1. Kernighan, Brian W., and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall Inc, Englewood Cliffs, NJ, 1978.
2. Johnson, S. C., "Lint, A C Program Checker," in *UNIX Programmer's Manual: Seventh Edition, Volume 2*, ed. Bell Telephone Laboratories, Holt, Rinehart and Winston, 1982
3. Robson, David, "Object-Oriented Software Systems," in the special Smalltalk-80 issue of *BYTE*, vol. 6, no. 8, August 1981, pp. 74-86.
4. Parnas, D. L., "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, December 1972.
5. Stroustrup, Bjarne, "Classes: An Abstract Data Type Facility for the C Language," *ACM SIGPLAN Notices*, vol. 17, no. 1, January 1982, pp. 42-52.
6. Stroustrup, Bjarne, "Data Abstraction in C," *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, part 2, October 1984, pp. 1701-1732.
7. Booch, Grady, "Object-Oriented Development," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, February 1986, pp. 211-221.
8. Liskov, Barbara, and Stephen Zilles, "Programming with Abstract Data Types," Proceedings of ACM Conference on Very High Level Languages, in *ACM SIGPLAN Notices*, vol. 9, no. 4, April 1974, pp. 50-59.
9. LaLonde, Wilf R., and John R. Pugh, "Specialization, Generalization, and Inheritance: Teaching Objectives Beyond Data Structures and Data Types," *ACM SIGPLAN Notices*, vol 20, no 8, August 1985.
10. Parnas, David L., "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1, March 1976.
11. Stroustrup, Bjarne, "An Overview of C++," *ACM SIGPLAN Notices*, vol. 21, no. 10, October 1986, pp. 7-18.
12. *Draft Proposed American National Standard for Information Systems - Programming Language C*, X3J11, October 1, 1986.
13. Stroustrup, Bjarne, *The C++ Programming Language*, Addison-Wesley, 1986.
14. Stroustrup, Bjarne, "An Extensible I/O Facility for C++," *Proceedings of USENIX Summer 1985 Conference*, Portland, Oregon, pp. 57-70.
15. Kristol, David M., "Four Generations of Portable C Compiler," *USENIX Summer Conference Proceedings*, Atlanta, GA, 1986, pp. 335-343.
16. Wilson, Cindy and Leon J. Osterweil, "Omega - A Data Flow Analysis Tool for the C Programming Language," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 9, September 1985, pp. 832-838.
17. Cargill, Thomas A., "Pi: A Case Study in Object-Oriented Programming," in the *OOPSLA'86 Conference Proceedings*, ed. Norman Meyrowitz, *ACM SIGPLAN Notices*, vol. 21, no. 11, November 1986, pp. 350-360.
18. Steffen, Joseph L., "Interactive Examination of a C Program with Cscope," *USENIX Winter Conference Proceedings*, Dallas, 1985.

# Program Translation By Manipulating Abstract Syntax Trees

*Xing Liu*, Software Scientist

*Patrick Conley*, President

Abraxas Software Inc.  
7033 SW Macadam Ave.  
Portland, Oregon 97219

October 1987

## 1. Introduction

We describe *CPPC*, a c++ preprocessor that translates c++ source code into c source code. CPPC is one of a few serious attempts at implementing a PC version of the c++ programming language, an *object-oriented* programming language designed and implemented (on VAX/UNIX) by Stroustrup [Stro86]. CPPC intends to support full c++ syntax and semantics. It is therefore quite different in scale from some other similar implementation efforts such as that of MINIMAL-C++ [TeGo87].

There are sound reasons for doing language translations between high level languages. The most practical consideration for writing a translator from a high level language to another is that of availability of the source language facilities. For instance, one may want to run application programs written in a language that no tools readily available to translate the programs into their executable forms. Another argument is that of software quality, namely programs should be easily understandable thereby to shorten the development cycle and to minimize the maintenance cost. To this

end, some programmers may have a preference as to which language they use to write programs. Recent development in software world has shown an increasing interest in this type of language translators and there are quite a few of them that are commercially available.

Like current implementation, CPPC is being constructed as a c-preprocessor. However, our design decision of having the translator explicitly build a complete abstract syntax tree for the program being translated makes it quite flexible to switch to other approaches in the later stages (compiling into native machine codes, for example). Our architectural consideration for CPPC takes the advantage of the fact that the source language and the object language share a great number of syntactical similarities (in fact, the syntax of c++ is a *superset* of that of c). Our approach is novel in that we do the translation relying on manipulating abstract syntax trees. Our design of the name mapping from the c++ code to c code takes into consideration easy debugging, an issue that is not well addressed by existing c++ translators.

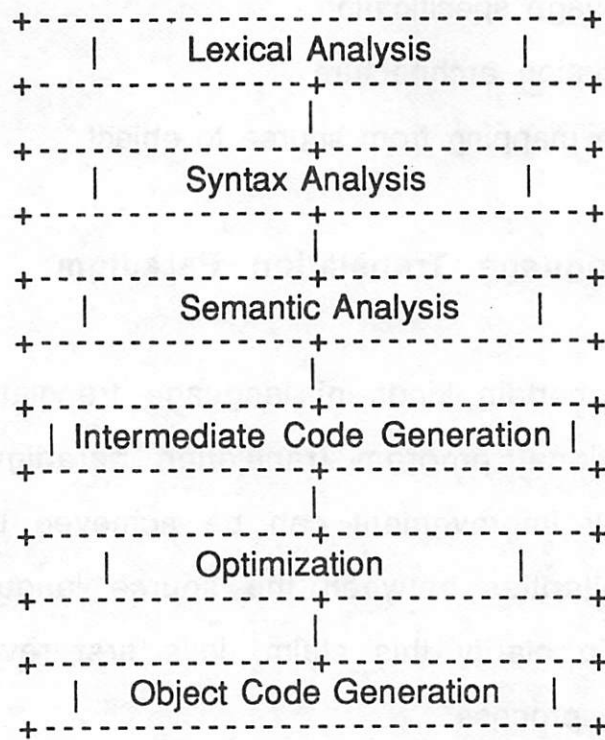
To reiterate, CPPC has the following features that distinguish it from other similar systems:

- 1). full c++ language specification
- 2). faster translation architecture
- 3). simpler name mapping from source to object

## 2. Traditional Language Translation Paradigm

Our theme is that certain kinds of language translators need not follow the conventional program translation paradigm. Moreover, significant efficiency improvement can be achieved by taking the advantages of similarities between the source language and the object language. To clarify this claim, let's first review a typical program translation process.

Program translation traditionally refers to a process through which computer routines in high level languages, such as FORTRAN and PASCAL, are converted into equivalent ones in low level languages, such as *assembly* languages and *machine* languages. It is such a well understood activity that architectures for this kind of translation systems have almost become standardized [AhUI77], as is depicted by the following diagram:



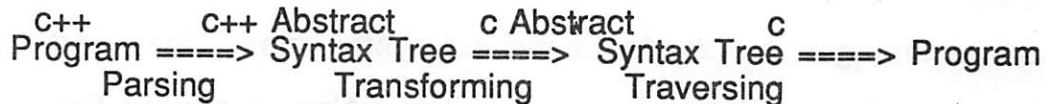
This multiple stage architecture is needed if the source languages differ from the object languages significantly both in syntax and in semantics. On the other hand, some of the stages may be bypassed if there are certain similarities shared by both the source language and the object language, to speedup the translation process.

Typically the output from the syntax analysis phase is a structure, called an *abstract syntax tree*, representing the source program being translated, and a *symbol table* that records attributes of variables appearing in the program. The symbol table, even though not shown in the diagram above, is a piece of important data structure that is used repeatedly throughout the translation process.

Therefore, the speed of the related table lookup routines are crucial to the efficiency of the translator.

### 3. The CPPC Approach

Schematically, the approach taken by CPPC for translating c++ programs into c programs can be illustrated as follows:



where an arrow  $\xrightarrow{\quad}$  indicates one step in the translation process, a starred arrow  $\xrightarrow{\quad}^*$  indicates zero or more applications of transformation on abstract syntax trees. In words, the translation process starts by parsing a c++ source program to generate an abstract syntax tree; zero or more transformations are then applied to this abstract syntax tree, resulting in another abstract syntax tree that represents the corresponding c program; finally, one pass traversal over the transformed tree generates desired c programs.

We believe this technique is applicable in most cases where translators among high level languages are to be built. The scheme can be advantageous over the traditional architecture in the following sense. First, similarities between the source language and the object language can be exploited to simplify the translation task. Second, unnecessary bookkeeping overhead is avoided since the amount of information that needs to be kept across the translation

phases is reduced, thus improving the efficiency of the translator. Third, the transformation rules can be formalized, one constructive step towards generating language translators automatically.

#### 4. Language Definition

In this section, we present detailed language definition for CPPC. The definition is largely based on the c++ reference manual [Stro86]. However, some changes are made to reduce syntax ambiguities and to ease the task of source program recognition. Also, the definition is given in a form that is an acceptable grammar specification directly executable by YACC, an LR(1) parser generator [John78].

```
prog
: defs
;

defs
: def
| defs def
;

def
: func_def
| data_decl
| type_decl
| func_decl
;
```

```

type_decl
: enum_spec ';'
| unio_spec ';'
| clas_spec ';'
;

func_def
: func_head func_body
;

func_decl
: func_head ';'
;

func_body
: comp_stmt
| mem_init_list comp_stmt
;

func_head
: sc_spec ft_spec tp_spec decl '(' arg_decl_list ')'
| sc_spec ft_spec decl '(' arg_decl_list ')'
| sc_spec tp_spec decl '(' arg_decl_list ')'
| sc_spec decl '(' arg_decl_list ')'
| ft_spec tp_spec decl '(' arg_decl_list ')'
| ft_spec decl '(' arg_decl_list ')'
| tp_spec decl '(' arg_decl_list ')'
| decl '(' arg_decl_list ')'
| sc_spec ft_spec tp_spec decl '(' ')'
| sc_spec ft_spec decl '(' ')'
| sc_spec tp_spec decl '(' ')'
| sc_spec decl '(' ')'
| ft_spec tp_spec decl '(' ')'
| ft_spec decl '(' ')'
| tp_spec decl '(' ')'
| decl '(' ')'
;

mem_init_list
: mem_init
| mem_init_list ',' mem_init
;

mem_init
: IDENTIFIER '(' expr ')'
| IDENTIFIER '(' ')'
;

```

```

data_decl
: sc_spec tp_spec decl_list ';'
| tp_spec decl_list ';'
| sc_spec decl_list ';'
| Typedef tp_spec decl_list ';'
;

```

```

tp_spec
: simp_tname
| clas_spec
| unio_spec
| enum_spec
| Const simp_tname
| Const clas_spec
| Const unio_spec
| Const enum_spec
;

```

```

simp_tname
: TYP
| Char
| Unsigned Char
| Short
| Unsigned Short
| Int
| Unsigned Int
| Long
| Unsigned Long
| Unsigned
| Float
| Double
| Void
;

```

```

enum_spec
: Enum IDENTIFIER '{' enum_list '}'
| Enum IDENTIFIER '{' enum_list '}'
| Enum IDENTIFIER
;

```

```

enum_list
: enumerator
| enum_list ',' enumerator
;

```

```

enumerator
: IDENTIFIER
| IDENTIFIER '=' const_expr
;

```

```

union_spec
: Union IDENTIFIER '{' defs '}'
| Union '{' defs '}'
| Union IDENTIFIER
;

class_spec
: class_head '{' defs '}'
| class_head '{' '}'
| class_head
| class_head '{' Public ':' '}'
| class_head '{' defs Public ':' defs '}'
| class_head '{' Public ':' defs '}'
| class_head '{' defs Public ':' '}'
;

class_head
: Struct IDENTIFIER
| Class IDENTIFIER
| Struct
| Class
| Struct IDENTIFIER ':' Public IDENTIFIER
| Struct ':' Public IDENTIFIER
| Struct ':' IDENTIFIER
| Class IDENTIFIER ':' Public IDENTIFIER
| Class ':' Public IDENTIFIER
| Class ':' IDENTIFIER
;

sc_spec
: Auto
| Extern
| Register
| Static
;

ft_spec
: Inline
| Overload
| Virtual
| Friend
;

decl_list
: init_decl
| decl '(' expr ')'
| decl_list ',' init_decl
| decl_list ',' decl '(' expr ')'
;

```

```

init_decl
: decl
| decl init
;

init
: '=' expr
| '=' init_list
;

init_list
: '{' expr '}'
| '{' expr ',' '}'
| '{' init_list ',' init_list '}'
;

decl
: dname
| '(' decl ')'
| '*' decl
| '&' decl
| decl '[' ']'
| decl '[' const_expr ']'
;

dname
: simp_dname
| IDENTIFIER DOUBLE_COLON simp_dname
;

simp_dname
: IDENTIFIER
| '~' IDENTIFIER
| operfunc_name
;

operfunc_name
: Operator op
;

arg_decl_list
: TRIPLE_DOT
| args
| args TRIPLE_DOT
;

args
: arg_decl
| args ',' arg_decl
;

```

```

arg_decl
: tp_spec decl
| tp_spec decl '=' expr
| type_name
;

```

```

comp_stmt
: '{' stmt_list '}'
| '{'
;

```

```

stmt_list
: stmt
| stmt_list stmt
;

```

```

stmt
: data_decl
| comp_stmt
| expr ';'
| If '(' expr ')' stmt
| If '(' expr ')' stmt Else stmt
| While '(' expr ')' stmt
| Do stmt While '(' expr ')' ';'
| For '(' stmt expr ';' expr ')' stmt
| For '(' stmt ';' expr ')' stmt
| For '(' stmt expr ';' ')' stmt
| For '(' stmt ';' ')' stmt
| Switch '(' expr ')' stmt
| Case const_expr ':' stmt
| Default ':' stmt
| Break ';'
| Continue ';'
| Return expr ';'
| Return
| Goto IDENTIFIER ';'
| IDENTIFIER ':' stmt
;

```

```

expr
: term
| expr '*' expr
| expr '/' expr
| expr '%' expr
| expr '+' expr
| expr '-' expr
| expr DOUBLE_LEFT_ANGLE expr
| expr DOUBLE_RIGHT_ANGLE expr
| expr '<' expr
| expr '>' expr
| expr GREATER_EQUAL expr
| expr LESS_EQUAL expr
| expr DOUBLE_EQUAL expr
| expr NOT_EQUAL expr
| expr '&' expr
| expr '^' expr
| expr '|' expr
| expr DOUBLE_AMPERSAND expr
| expr DOUBLE_VERTICAL_BAR expr
| expr '=' expr
| expr PLUS_EQUAL expr
| expr MINUS_EQUAL expr
| expr TIMES_EQUAL expr
| expr DIVIDE_EQUAL expr
| expr MOD_EQUAL expr
| expr EXOR_EQUAL expr
| expr AND_EQUAL expr
| expr OR_EQUAL expr
| expr LEFT_SHIFT_EQUAL expr
| expr RIGHT_SHIFT_EQUAL expr
| expr '?' expr ':' expr
| expr ',' expr
;

```

term

```
: prim_expr
| '*' term
| '&' term
| '+' term
| '-' term
| '~' term
| '!' term
| DOUBLE_PLUS term
| DOUBLE_MINUS term
| term DOUBLE_PLUS
| term DOUBLE_MINUS
| sizeof expr
| sizeof '(' type_name ')'
| '(' type_name ')' prim_expr
| simp_name '(' expr ')'
| New type_name '(' expr ')'
| New type_name
| New '(' type_name ')'
| Delete expr
| Delete '[' expr ']' prim_expr
```

prim\_expr

```
: id
| DOUBLE_COLON IDENTIFIER
| konst
| STRING
| This
| '(' expr ')'
| prim_expr '[' expr ']'
| prim_expr '{' expr '}'
| prim_expr '(' ')'
| prim_expr ':: id
| prim_expr POINTER id
```

id

```
: IDENTIFIER
| operfunc_name
| IDENTIFIER DOUBLE_COLON IDENTIFIER
| IDENTIFIER DOUBLE_COLON operfunc_name
```

```

op
: '*'
: '&'
: '+'
: '-'
: '~'
: '!'
: DOUBLE_PLUS
: DOUBLE_MINUS
: '/'
: '%'
: DOUBLE_LEFT_ANGLE
: DOUBLE_RIGHT_ANGLE
: '<'
: '>'
: GREATER_EQUAL
: LESS_EQUAL
: DOUBLE_EQUAL
: NOT_EQUAL
: '^'
: '|'
: DOUBLE_AMPERSAND
: DOUBLE_VERTICAL_BAR
: '='
: PLUS_EQUAL
: MINUS_EQUAL
: TIMES_EQUAL
: DIVIDE_EQUAL
: MOD_EQUAL
: EXOR_EQUAL
: AND_EQUAL
: OR_EQUAL
: LEFT_SHIFT_EQUAL
: RIGHT_SHIFT_EQUAL
: '(' ')'
: '[' ']'
: New
: Delete
:

```

```

type_name
: tp_spec
| tp_spec abstract_decl
:

```

```

abstract_decl
: '('
| '(' arg_decl_list ')'
| '[' expr ']'
| '*' abstract_decl
| abstract_decl '(' ')'
| abstract_decl '(' arg_decl_list ')'
| abstract_decl '[' expr ']'
| abstract_decl '[' ']'
;

```

```

const_expr
: konst
;

```

```

konst
: I_CONSTANT
| C_CONSTANT
| F_CONSTANT
;

```

## 5. Current Status

As of the time of this writing, the parser for CPPC has been completed. Correct syntax trees have been produced and three transformation rules on syntax trees are being defined.

These transformations rules, implemented as a collection of mutually recursive routines, will transform abstract syntax trees from one form into another when applied to them. Specifically, the *type-lifting-rule* will eliminate nested class and struct definitions. The *aggregate-lifting-rule* will move in-place aggregate definitions acting as type specifiers to appropriate enclosing context, while leaving behind proper type names as needed. The *function-lifting-rule* will separate the structure definition from its behavior definition of a class, by lifting member function definitions out of

the class definition after proper mechanisms are added to resolve symbol name conflicts.

## 6. References

[AhUl77] Principles of Compiler Design, A.V. Aho and J.D. Ullman, Addison-Wesley, 1977.

[John78] YACC: Yet Another Compiler Compiler, Stephen C. Johnson, UNIX Programmer's Manual, Vol.II, 1978.

[Stro86] The C++ Programming Language, Bjarne Stroustrup, Addison-Wesley, 1986.

[TeGo87] MINIMAL C++ Report, Larry Tesler and David Goldsmith, Apple Technical Report No.2, 1987.

Thinking Machines Technical Report PL87-5

## **C\*: An Extended C Language for Data Parallel Programming**

**John R. Rose  
Guy L. Steele Jr.**

**April 1987**

## Abstract

C\* is an extension of the C programming language designed to support programming in the data parallel style, in which the programmer writes code as if a processor were associated with every data element. C\* features a single new data type (based on classes in C++), a synchronous execution model, and a minimal number of extensions to C statement and expression syntax. Rather than introducing a plethora of new language constructs to express parallelism, C\* relies on existing C operators, applied to parallel data, to express such notions as broadcasting, reduction, and interprocessor communication in both regular and irregular patterns. We discuss the goals of the language design, the specific realization of those goals, and the implementation of C\* on the Connection Machine® computer system.

© 1987 Thinking Machines Corporation

Connection Machine is a registered trademark, and

C\* is a trademark, of Thinking Machines Corporation.

VAX is a registered trademark of Digital Equipment Corporation.

Symbolics 3600 is a trademark of Symbolics, Inc.

## 1 Introduction

As large scale computer users tackle ever-larger problems, we often find that the amount of data to be processed is growing at a much faster rate than the code that directs the processing. It is not unusual now for an application to involve the handling of  $10^9$  data values. For such applications there is much more to be gained by exploiting parallelism in the data than parallelism in the code. Pipelined vector-processing computers, for example, while not completely parallel in their operation, implicitly take advantage of the same independence of data elements that makes possible true parallel computation.

If the operations performed were different for each data element, the programming situation would be intractable. Fortunately, this is not the case. Data intensive applications typically involve repeating the *same* processes, perhaps with minor variations, across *different* elements of data. The central insight of *data parallel programming* is the realization that a single program can operate on all the data at once, and that it is useful to organize and code algorithms accordingly.

A new style of programming and of programming language design is now emerging, the *data parallel style*, that allows programmers to express the processing of large amounts of data. This is really an old style in some ways; APL [7,6,3] allowed the expression of data parallel algorithms more than 20 years ago. However, two recent developments give the style new significance:

1. The availability of computers that have been purposely designed to support the data parallel style.
2. The integration of the data parallel style with the features of familiar sequential programming languages.

An important advantage of the data parallel style is that the parallelism in the data usually grows as the amount of data grows, whereas the amount of control parallelism may remain fixed. This is not to say that control parallelism is to be ignored, but simply that programs and computers organized to exploit data parallelism are in a better position to provide increasingly good parallel performance for large-scale problems.

Data parallel computer architectures take advantage of the parallelism that exists in the data of nearly all large-scale applications. An operation that needs to be done on one data element almost always needs to be done for many other data elements as well, and (almost always) they can all be done at the same time. Data parallel computers do this by, in effect, assigning a separate processor to each element of data. In practice this may be implemented by multiple physical processors, timeslicing techniques (possibly pipelined), or a combination of techniques. Nevertheless, the power of the style arises from providing the programmer with the illusion (not necessarily a false one) that there are plenty of processors to go around.

C\* is a programming language with two major design goals, each with several important consequences and corollaries:

- To support the data parallel programming style.

- Easy to express parallel and cooperative operations on large amounts of data.
- Easy to express various patterns of communication (information flow) among data elements.
- Supports both a local and a global perspective of data; that is, one may write code as if operating on a single data element (and the code is then applied to all elements in parallel) or as if acting on entire arrays.
- To be compatible, both semantically and culturally, with a widely-used existing sequential programming language, in this case C [8,4].
  - The extended language should be a strict superset of the standard sequential language.
  - Extensions should be similar in feel and use to standard language features.
  - The extensions should introduce as few new concepts and constructs as possible.
  - The language should be portable and yet remain close to the machine, in that code compiles to simple, predictable machine operations. The point is to allow the programmer access to actual data parallel hardware, rather than to simulate some interesting but impractical abstract machine.
  - Where possible, extensions should be based on other well-known extensions to the C language rather than being invented from scratch.
  - The extended language should permit the use of a familiar program development environment and debugging tools.

An additional goal, in order to make it easier to reason about and to debug parallel programs, was that a program's behavior should be deterministic, predictable, and repeatable.

C\* supports the data parallel programming style while making minimal extensions to the C language. One new data type is introduced, and not a very new one at that, for it is based on the syntax and semantics of classes in C++ [11] an existing extended dialect of C for object-oriented programming. One new statement type is introduced, for initiation of parallel execution. A few new operators are introduced, but purely for reasons of convenience, rather than for supporting parallelism in a fundamental way. While C\* effectively allows the processing of large arrays of data, it preserves the interchangeability of arrays with pointers, a feature central to the C language. Indeed, far from finding pointers difficult to handle, C\* relies on pointers for interprocessor communication. C\* allows data parallel programming from the local perspective; a cardinal rule of the language is that when processors are not interacting they behave exactly as if each is executing an ordinary sequential C program.

In the remainder of the paper, we first present an overview of classes in C++ as background to some of the ideas in C\*. We then discuss the abstract machine model that underlies C\* and its mapping onto a specific computer architecture, the Connection Machine computer system. Memory layouts are then examined, followed by an extensive presentation of the treatment of expressions and statements in C\*. An example program is exhibited, followed by comparisons with other parallel C designs.

## 2 Overview of Classes in C++

The *class* concept in C++ is an extension of the *struct* (record type) concept in C. A class definition looks just like a struct definition, with these differences:

- The keyword `class` replaces the keyword `struct`.
- A class may have functions as members.
- A class may declare some members to be public, and others not; functions that are not declared to be part of the class (either as members or as "friends") may access only public members directly.

The separation of members into public and private categories allows information hiding and the construction of abstract data types. Combining this with the concept of member functions gives the language an object-oriented flavor; rather than accessing members directly, outside code must "send a message" to a class object by calling one of its member functions. (Such a member function must itself be public, of course.) In C++, a struct is just the special case of a class that declares all of its members to be public. If a struct also has no function members, then it is identical in all respects to an ordinary C struct.

Here is an example of a class definition and a declaration of some instances of the class.

```
class employee {
    double salary;
    employee_type type;
public:
    char *name;
    int knowledge;
    employee_type get_type();
    void raise(double);
    void print();
};
```

```
employee Fred, Sally, programmers[20];
```

Here there are four data members, of which only two, `name` and `knowledge`, are public. There are also three function members, all of which are public. Here are definitions for the three function members:

```
employee_type employee::get_type() {
    if (type == Secret_Agent)
        return Desk_Clerk;
    else return type;
}
```

```

void employee::raise(double amount) {
    if (amount >= 0)
        salary += amount;
    else Error("Bad raise");
}

```

```

void employee::print() { printf("%s", name); }

```

The public function `get_type` in effect allows other code to read the `type` of an employee, but prevents other code from writing it. This is more flexible than simply declaring the `type` component to be `const`, because one might want to allow a member function to write the `type`, but not a non-member function. Furthermore, `get_type` has a chance to “sanitize” the result, thereby hiding certain information.

Similarly, the member function `raise` provides protected access to the `salary` component; no one outside the class may read the `salary`, and anyone may increase it but no one may decrease it. The call

```
Sally.raise(5000.0);
```

gives Sally a much-deserved raise. Note that a member function is referred to using the same operators (`.` and `->`) as for data members.

What the member function `print` does, any non-member function could perform, for the `name` component is public. This function has been included to illustrate *overloading*. One can have a function named `print` for the class `employee` and another named `print` for another class, say `part`, for after all every class, like every struct, has a separate namespace for names of members.

```

class part {
    int part_number;
    double price;
    vendor *supplier;
    char *description;
    void print();
};

```

```

void part::print() {
    printf("%d [%s]", part_number, description);
}

```

One may therefore write

```
Flange.print();
```

and the appropriate `print` function will be used, depending on whether `Flange` is an aluminum flange or George Flange. Built-in operators such as `+` and `*` are initially overloaded

so as to operate on both integers and floating-point numbers; the user may additionally overload them, in the same manner as for user-defined functions, to handle user-defined class types.

Note that a member function can be called only by first computing some particular object that is an instance of that class, and then calling the function by selecting it as a member of that instance. Within the code of a member function, the magic variable `this` always has as its value a pointer to the very class instance for which the member function was invoked. The code of a member function may also refer by name to any member of the class, and the effect is to refer to that member within the invoking class instance. It follows that within the function `raise`, for example, referring to `salary` means exactly the same thing as referring to `this->salary`.

The design of C++ encourages the programmer to work in terms of local operations on a single datum at a time. Naturally, defining a C++ class includes specifying its interface (via public members) to the rest of the program. But with that done, the programmer is free to adopt a local perspective: Each instance of a class is a tiny program, with its own data set (the member variables) and code (the member functions). The scoping rules of C++ are such that, within this microcosm, the member variables and functions are referred to in the same way as global variables and functions.

Given a program that mostly performs local operations inside member functions, and an array of many class instances, it is often only an accident of implementation that the program's execution operates on each instance sequentially; parallel execution would do just as well. The parallel mode of operation is what C\* provides to the programmer.

### 3 Data Parallel Machine Model

Just as the C language assumes an abstract machine model with certain interesting abstract properties (sequential execution, uniform address space, meaningful pointer arithmetic), so C\* assumes a certain abstract machine model. The C\* model is an extension of the plain C model. They share such important features as a uniform address space and meaningful pointer arithmetic. C\* extends C by having many processors instead of just one, all executing the same instruction stream. The C\* model may be summarized as providing the programmer with *lots of processors* of an otherwise *conventional* nature, operating within a *uniform address space* in a *synchronous execution* mode. These four points deserve additional comments:

- **Lots of processors.** How many is lots? A thousand? A million? The answer is "enough." Enough for what? Enough that a processor can be assigned to every data item of interest. Depending on the problem, that could be hundreds or billions. (This is analogous to ordinary C assuming "lots of memory," that is, "enough.") This characteristic is important because it allows the model to scale well. A well-written sequential C program using arrays can process ten times as much data as before if the arrays are declared to be ten times as big, without changing any other part of the program, provided only that the hardware on which it runs can provide

the required memory resources. C\* programs can scale well only if the programmer can assume that there will be enough processors as well as enough memory.

- **Conventional processors.** What is conventional? A VAX? A Motorola 68000? A National Semiconductor 32032? We take "conventional" to mean "can execute ordinary sequential C code." One consequence is that each processor has some amount of memory associated with it.
- **Uniform address space.** This characteristic and the uses of pointers that it makes possible are so close to the heart of C that no parallel version of C can abandon it without giving up its claim to be a superset of C. Therefore in effect *any* processor can access *any* part of memory. (The means by which this is implemented may vary, of course.) This implies that communication among processors cannot be limited to such fixed patterns as two-dimensional grids (although the model says nothing about whether or not such special cases might be faster than the general case).
- **Synchronous execution.** C\* assumes a synchronous model of computation, in which all instructions are issued from a single source, a distinguished processor called the *front end*. All the other processors constitute the *processor array*, and are called *data processors*. At any time, the data processors that are executing the instruction stream sent out from the front end are called the "active set." The local memory of an idle processor does not change, unless another processor writes it. (This model of computation may appear to be inherently SIMD. As will be seen, however, the model does allow code with MIMD-like, but nevertheless synchronous, behavior.)

Figure 1 shows the architecture of the Connection Machine computer system [5] and how it maps onto the abstract C\* machine model. The front end processor, with its associated memory, is in fact a conventional computer, such as a Symbolics 3600 Lisp machine or a VAX. The data processors are of a proprietary but fairly straightforward design; they can carry out all ordinary C data operations. The current largest Connection Machine system configuration has 65,536 (that is,  $2^{16}$ ) data processors. Each data processor has 8K bytes of local memory.

All instructions, whether for the front end or for the data processors, reside in the memory of the front end. Front end code is executed by the front end in the usual manner. The Connection Machine processor array in effect extends the instruction set of the front end; parallel instructions are "fetched" by the front end, but rather than being executed directly are broadcast through a special instruction bus from the front end to the data processors. This bus may also be used to broadcast data to the data processors (in effect by making it be the immediate operand of a "move-immediate" instruction).

Data may reside in the front end memory or in the memories of the data processors. There is a data bus that allows the front end to access the memories of the data processors a word at a time, for both reading and writing; in effect, the local memories are in the address space of the front end computer.

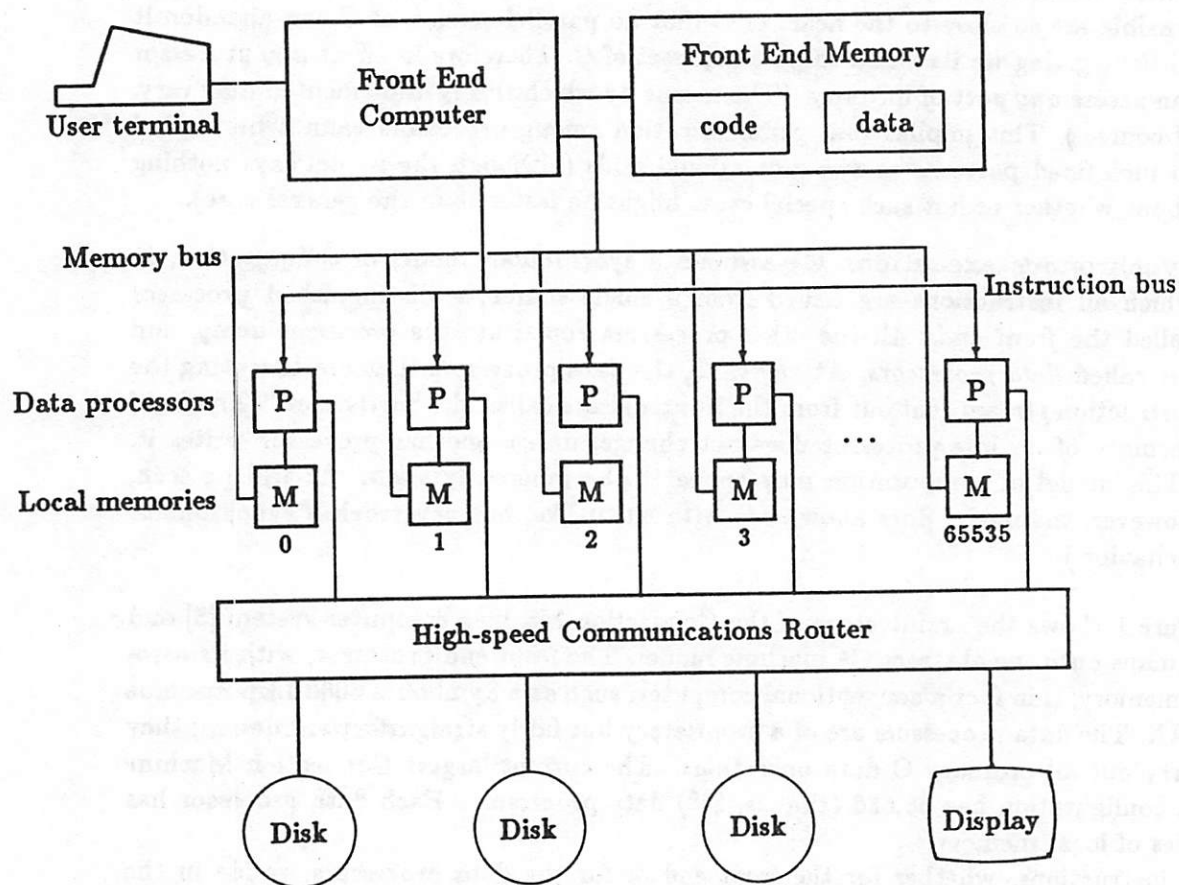


Figure 1: Connection Machine Architecture

There is a high-speed communications device (it would be misleading to call it either a bus or a network), the *router*, that interconnects all of the data processors. It allows each processor to send a message to any other processor, all at the same time, with a total throughput exceeding 3 gigabits per second.

(The ability to do I/O, although irrelevant to the abstract C\* machine model, is of course important architecturally and functionally. Figure 1 illustrates the fact that I/O may be done either through the front end in the conventional manner or through a high-bandwidth bus into the router.)

Although 65,536 may seem like a large number of processors, frequently it is not enough. Furthermore, many data parallel applications don't need as much as 8K bytes of memory per data processor. The idea in data parallel programming is to assign a processor to each data structure of interest, and it is a rare C program in which a `struct` occupies as much as 8K bytes! Using a processor with 8K bytes of memory to hold a sixteen-byte structure may strike one as a rather extravagant use of memory.

The Connection Machine system solves these problems by supporting *virtual processors*. The system logically divides the memory of each data processor into  $n$  equal portions, and processes each directive  $n$  times, once for each portion, thereby timeslicing the physical processors  $n$ -fold. (This description glosses over many interesting details, especially the special hardware support in the router for communication among virtual processors.) The overhead for this timeslicing is negligible, and provides to the front end the illusion of having  $n \times 65,536$  data processors, each with  $8K/n$  bytes of memory and  $1/n$  the speed of a physical processor. The virtual processing ratio  $n$  may be set under software control; the following table illustrates some possible configurations.

$n$	Virtual processors	Memory each
1	64K	8K bytes
2	128K	4K bytes
4	256K	2K bytes
8	512K	1K bytes
16	1M	512 bytes
32	2M	256 bytes
64	4M	128 bytes
128	8M	64 bytes
256	16M	32 bytes
512	32M	16 bytes
1K	64M	8 bytes
2K	128M	4 bytes

The virtual processor mechanism is supported in firmware. Once the front end has established the ratio  $n$ , it need not be further concerned with the fact that virtual processors are in use. This supports the scalability of C\* programs in two important ways:

- As the problem size increases, requiring more virtual processors, the same set of physical processors can run the same code, provided that there is enough total

memory, and at a slower rate per virtual processor, the total throughput remaining the same.

- As technology improves, allowing new hardware with more physical processors, the same code with the same number of virtual processors can be run faster, with fewer virtual processors per physical processor, provided that the problem uses at least as many virtual processors as the new number of physical processors.

We emphasize the advantage of not having to change the code (except perhaps to redeclare array sizes) to take advantage of these two scaling properties. Because of the flexibility of the virtual processor mechanism, only the total amount of memory, not the number of physical processors, constrains how large a problem may be tackled.

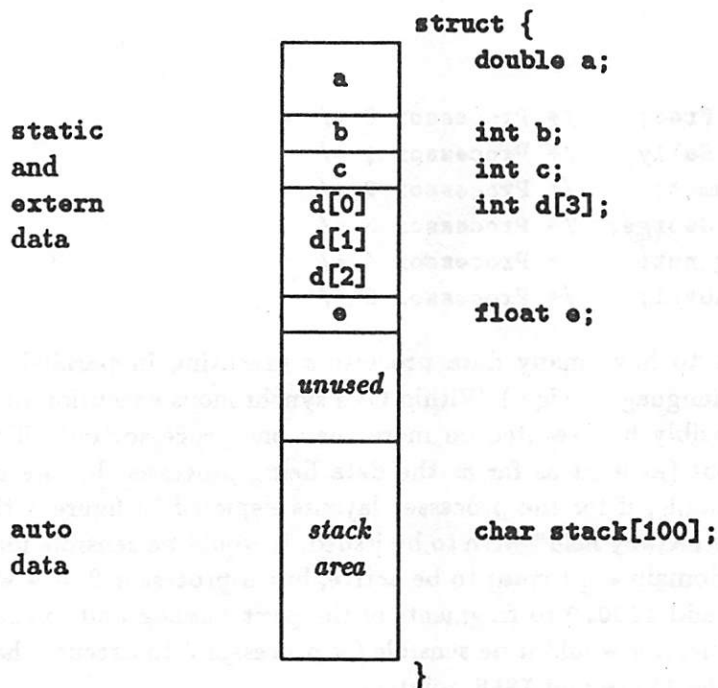
### 4 Processor Memory Layouts

The layout of memory within each data processor is conventional. Except for the fact that no code is stored in the memory of a data processor, memory is laid out exactly as for a C program in a conventional sequential computer. One end of memory is used to hold statically allocated variables (storage classes `static` and `extern`), and the other end is used as a stack area for the allocation of automatic variables (storage class `auto`).

Processor memory layout can be informally described as a record structure, that is, a C struct. (The C language is very good at describing arbitrary memory layouts.) See figure 2.

When there are many processors, as in the C\* machine model, different processors may have different memory layouts because they may hold different kinds of data for different purposes. In figure 3 are shown six hypothetical memory layouts for six processors. (The stack area is relevant, but for present purposes is not shown in the figure.) Processors 0, 1, and 3 happen to be laid out in the same way, but processors 2 and 4 are laid out in another way, and processor 5 in yet another manner. If we think of a data processor's memory layout as being a record structure, then we might as well say that a processor's memory really *does* belong to such a structure type, and we can distinguish groups of processors by that type. In C\* a structure type that describes the memory of a data processor is called a domain. The processor layouts in figure 3 may be described in C\* as follows:

```
domain employee {
    double salary;
    employee_type type;
    char *name;
    int knowledge;
};
```



**Figure 2: Example Memory Layout in a Data Processor**

```

domain part {
    int part_number;
    double price;
    domain vendor *supplier;
    char *description;
};

```

```

domain book {
    char *title, *ISBN;
    int content;
    domain employee *owner;
};

```

```

domain employee Fred;    /* Processor 0 */
domain employee Sally;   /* Processor 1 */
domain part grommet;     /* Processor 2 */
domain employee George;  /* Processor 3 */
domain part wing_nut;    /* Processor 4 */
domain book my_novel;    /* Processor 5 */

```

In C\* we want to have many data processors executing in parallel. (That is the whole point of the language design.) Within C\*'s synchronous execution model, the same instruction can sensibly be executed on more than one processor only if they have the same memory layout (at least as far as the data being processed by the instructions in question). For example, if for the processor layouts depicted in figure 3 the instruction "add 1000.0 to the salary field" were to be issued, it would be sensible for processors 0, 1, and 3 (those of domain `employee`) to be active, but if processor 2 or 4 were active the result would be to add 1000.0 to fragments of the `part_number` and `price` fields, which would not be sensible; nor would it be sensible for processor 5 to execute that instruction, thereby mangling the `title` and `ISBN` pointers.

To state the principle generally: it is sensible for many processors to execute the same instructions only if they are of the same type, that is, they belong to the same domain. This is a fundamental aspect of the design of C\*. Indeed, the name scoping rules of C\* are arranged so as to enforce this; it is simply impossible to write code in C\* that violates the principle.

Where have we seen before the idea of code that can be executed only for a specific data type? Member functions in C++. It is no accident that the declaration syntax for a C\* domain is the same as that of a C++ class (except for the particular keyword used). In accordance with the design goal of using familiar language features in preference to inventing new ones, we have adopted the syntax and semantics of C++ for C\* wherever relevant. (Not all features of C++ have been taken over, however; we discuss this point

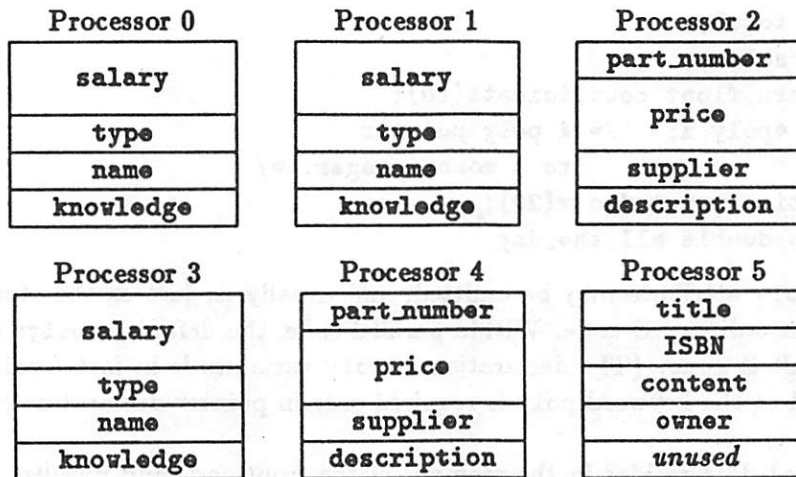


Figure 3: Example Memory Layouts in Several Data Processors

further in section 10.) So, for example, one may define a function that is a member of a domain:

```
void employee::raise(double amount) {
    if (amount >= 0)
        salary += amount;
    else Error("Bad raise");
}
```

This code is character-for-character identical to the C++ function of the same name shown in section 2. The semantic difference is that in C\* this function may be called on behalf of many employees simultaneously, and they will all be given raises in parallel.

In C\*, all code is divided into two kinds: serial and parallel. Code that belongs to a domain is parallel, and may be executed by many data processors at once. All other code is serial, and is executed by the front end as if it were ordinary sequential C code. The two types of code are distinguished by syntactic context: code may belong to a domain (and therefore be parallel) only as the body of a member function of the domain or as the substatement of a selection statement (discussed later in this paper) that selects the domain. Once the context is established, however, the two types of code are written using the same syntax; parallel code, taken out of context, looks exactly like ordinary sequential C code.

In C\*, all data is also divided into two kinds: scalar and parallel. These are described in the language using two new keywords, *mono* and *poly*; they are used somewhat like the storage class keywords *extern*, *static*, and *auto*, but describe an independent attribute. In certain situations they may sensibly be used in the same way as the *const* and *volatile* keywords of proposed ANSI standard C. Some example declarations:

## §5 Parallel Expressions

```
mono int total;
poly int salary;
poly extern float coefficients[10];
mono int *poly x; /* A poly pointer
                  to a mono integer. */
poly static struct foo x[20];
poly auto double all_the_day;
```

The mono or poly attribute may be omitted, and usually is, just as the storage class is often omitted in ordinary C code. Within parallel code, the default is poly; within serial code, the default is mono. (The declaration of poly variables is in fact forbidden within serial code, and so the keyword poly is required only in pointer-declaration contexts and casts.)

Scalar (mono) data resides in the memory of the front end, and parallel (poly) data resides in the memory of the data processors. Note that poly data is only *potentially* parallel; it is processed in parallel only if referred to by parallel code. It is possible for the front end, executing serial code, to access poly data in a sequential manner. Similarly, serial data may be processed by many data processors at once if the front end will first broadcast copies.

Domains differ from classes in that member declarations for domains can use the storage class keywords `auto`, `register`, `static`, and `extern`. In particular, different files can declare different members of a domain, and the `extern` keyword can mark members that are defined in one file but referenced in another. (In contrast, a C++ class may not be declared in such a piecemeal fashion.) Note that `auto` variables in member functions are allocated within each instance, on per-processor stacks. This is all consistent with the fact that the memory of each data processor is organized in the same way as for a sequential C program.

## 5 Parallel Expressions

### 5.1 New Operators

We would like to be able to say that we didn't add any new operators to the language. The truth is that we did, but for reasons not directly related to supporting parallelism.

First, we added maximum and minimum operators, which are really arithmetic operators. The minimum operator `<?` and the maximum operator `>?` group left-to-right; their precedence is lower than the shift operators `>>` and `<<` and higher than the relational operators. They may be understood in terms of their traditional macro definitions

```
a <? b => ((a) < (b)) ? (a) : (b)
a >? b => ((a) > (b)) ? (a) : (b)
```

but of course the operators, unlike the macro definitions, evaluate each argument exactly once. The operators `<?` and `>?` are intended as mnemonic reminders of these definitions. (Such mnemonics are important. The original design of C\* used `><` and `<>` for the

maximum and minimum operators. We quickly discovered that users had some difficulty remembering which was which.) Note that these operators may be applied to pointers as well as to numeric data. By themselves these operators are relatively unimportant (and indeed are normally expressed as macros in ordinary C rather than being built in), but the assignment operators `<?=` and `>?=` have great utility in C\*, as can be seen in examples below (see section 5.3).

Second, in C\* most assignment operators may be used as unary operators. As we will see, this unary use of existing binary operators is introduced purely for convenience, as an abbreviation for a frequently used and otherwise rather awkward idiom.

## 5.2 Patterns of Communication

Instead of adding new operators for parallel computation, C\* takes advantage of the compile-time type distinction between scalar (mono) and parallel (poly) data, and extends existing operators, through overloading, to operate on parallel data. These extended interpretations allow us to express various interesting patterns of communication:

- *reading*: fetching one value from a particular data processor to the front end
- *writing*: storing one value from the front end into a particular data processor
- *replication*: broadcasting a value from the front end to all parallel processors
- *reduction*: reading values from all parallel processors, delivering a combined result
- *permutation*: communication among data processors (in both regular and irregular patterns)

We achieve all of these patterns of communication by using the standard C operators and adding two rules to the usual rules of C evaluation:

**Replication Rule:** A scalar value is automatically replicated where necessary to form a parallel value.

**As-If-Serial Rule:** A parallel operator is executed for all active processors *as if* in some serial order.

The Replication Rule requires that when a binary (or ternary) operator combines mono and poly data, the mono value is replicated before you do the operation. A mono value is also replicated if passed as an argument to a function whose corresponding formal parameter is poly. In other words, *replication* occurs wherever necessary, by fiat. This is a fairly pedestrian rule, found in almost every parallel programming language (consider, for example, Fortran 8x [13], PASM Parallel-C [9], and APL [7,6,3] as representative examples).

In the following example, `increase` is a mono variable, residing in the front end. In the assignment statement all employees are given a raise of the same amount, because the value of `increase` is replicated before being added to the individual salary components.

## §5 Parallel Expressions

```
double increase = 178.43;
...
[domain employee].{ salary += increase; }
```

The As-If-Serial Rule is rather devious, because it facilitates parallelism by imposing a sequential semantics! When an operator operates on poly values, it is *as if* the active processors execute the operator by taking turns in some (unspecified and unpredictable) serial order.

Why have we adopted this to all appearances so completely bizarre rule? Surely it makes no difference whether a thousand additions are done sequentially or all at once! That is true, but it *does* make a difference whether *assignment* operators are done sequentially or all at once, because they have side effects. (This is also true of the unary *increment* and *decrement* operators.) To see why, we must return to the Replication Rule, and realize that when a mono lvalue is replicated, it is replicated *as an lvalue*. Consider the following code:

```
double total_salary;
...
{
    total_salary = 0;
    [domain employee].{
        total_salary += salary;
    }
}
```

The second assignment statement will first replicate the variable `total_salary` as an lvalue; then the processor for every employee will attempt to perform the `+=` operation on its own salary and that same lvalue. The As-If-Serial Rule is a simple way of stating the guarantee that the processors in effect do not interfere with each other. Without such a guarantee one might imagine scenarios where several processors read `total_salary`, then all perform the addition, then all write their sums back, with the result that some values are lost from the total. The semantics of C\* state that when the dust clears every processor's salary value has been added into `total_salary` exactly once. (We make a few more remarks about the As-If-Serial Rule below in conjunction with the discussion of *permutation*.)

This, then, is how *reduction* is expressed in C\*. The other C assignment operators may be used in a similar manner. For example, this code fragment performs AND-reduction:

```
mono unsigned combined_mask = -1;
...
combined_mask &= mask;
```

(Here assume `mask` to be a poly variable.) Note that the variable `combined_mask` must be initialized to `-1` and not `0`, because `-1` is the identity value for bitwise AND operations.

The code in both the previous examples initializes the mono collecting variable before performing the combining assignment. Without this initialization, the variable would receive the result of combining all the parallel values *as well as* its own previous value. Sometimes this is useful. However, the pattern in the examples—initializing a mono variable to an identity value and then using a compound assignment to combine a poly value—is so common that C\* provides an abbreviation. One may use any assignment operator except `=`, `%=`, `<%=`, or `>%=` as a unary operator. Its operand must be of poly type, and the value is the mono result of combining all the value of the poly type. Thus we have:

```

+=x   sum of the active x values
*=x   product
<?=x  minimum
>?=x  maximum
|=x   logical OR
&=x   logical AND
^=x   logical EXCLUSIVE OR
-=x   negative of the sum
/=x   reciprocal of the product

```

For example, to find the largest salary within the domain of employees, one need only write `>?= salary`; here we see why the maximum-assignment and minimum-assignment operators are so useful in C\*. To find the average salary, one may write

```
(+= salary) / (+= (poly)1)
```

which sums the salaries, counts the number of employees (by adding up one 1 for each active processor), and then takes the quotient; this example has a somewhat APL-like flavor to it. To obtain a pointer to the lowest-numbered active processor, the expression `<?= this` suffices.

To summarize the discussion of expressions up to this point: *replication* is built into the language in the usual manner, and *reduction* is expressed using the assignment operators. The other three patterns of communication, namely *reading*, *writing*, and *permutation*, arise naturally from the fact that addressing and the use of pointers in C\* is perfectly as general as in C. To put it another way, the language restrictions that you might fear would be imposed because of implementation considerations are *not* imposed after all. This does make work for an implementor of the C\* language; such language constructs as pointer indirection may be implemented in several wildly different ways, depending on the data types involved. The point, however, is to provide a clean and complete programming model to the user.

The communication pattern of *reading* is expressed quite simply. Within serial code one might write, for example,

```
strcmp(programmer[2].name, "George Flange");
```

Recall that `programmer` is an array of twenty employees, and so the elements of this array are instances of the class `employee`, residing in the memory of the processor array. The front end can refer to the name component of programmer number 2 simply by referring to `programmer[2].name` in the natural way.

*Writing* is expressed in exactly the same manner; for example, because the name component is public and writable, one can change an employee's name in the obvious way:

```
programmer[2].name = "George Bushing";
```

*Permutation* is also achieved through the natural use of C pointers. There is no reason why one parallel processor cannot have a pointer into the memory of another processor. Therefore if `x` is some poly variable of type `T`, say, and `p` is a poly variable of type "pointer to `T`," then the statement

```
*p = x;
```

means "send message `x` to processor `p`" (or more precisely to a specific variable within a processor, both being indicated by `p`), and all active processors can do this in parallel.

The use of an explicit pointer variable `p` allows any topological communications pattern to be expressed. The space required for such a pointer may be eliminated in cases where the pattern is sufficiently regular that it may easily be computed "on the fly." Here the ability of the C language to express address arithmetic comes in handy; every processor can obtain a pointer to itself (by referring to the variable `this`) and then perform arithmetic on that pointer, allowing all kinds of relative addressing. For example,

```
x = (this+1)->x;
```

causes all `x` values to be shifted downward by one processor (every processor fetched the `x` value from the processor one above it).

### 5.3 Examples of Communication

Here is a simple example. Recall that every book has, among other components, an owner that is an `employee`:

```
domain book {
    char *title, *ISBN;
    int content;
    employee *owner;
    void print();
};
```

We can print the number of books whose owners earn less than 40000.0 in this way:

```
[domain book].{
    if (owner->salary < 40000.0)
        printf("%d books\n", +=(poly)1);
}
```

Note that at most one call occurs to `printf`, not one for each data processor, because in this call all its arguments are `mono`. The interesting part here is the expression `owner->salary`. This is executed by the processors for all the books; every book processor fetches the `salary` component that is stored in the memory of the `employee` processor for that book's owner. All the book processors can do this in parallel, and they can then all compare the fetched values to `40000.0` in parallel. (There is also a minor bug in this example, which is that if *no* book owner earns less than `40000.0` then nothing is printed. We return to this point below in section 6.2, in the discussion of the semantics of the `if` statement.)

Here is another example of interprocessor communication. Under the (perhaps fanciful) assumption that owning a book increases its owner's knowledge by a certain amount, we can write this code:

```
[domain book].{ owner->knowledge += content; }
```

In this case the processor for every book will cause the `content` of that book to be added to the `knowledge` component of its owner. If several books have the same owner, then that owner's `knowledge` will be increased by the combined content of all of the books. In short, during communication expressed through pointer indirection, the mapping from active processors to accessed processors need not be one-to-one. The C\* language supports not just *permutation*, but in fact *multiple parallel broadcasts* (by parallel fetches through pointers) and *multiple parallel reductions* (by parallel stores through pointers using C assignment operators). Indeed, multiple broadcasts and multiple reductions may occur in a single statement (for example, `*q += *p;`).

As a large example illustrating various patterns of communication, here is a bit of code that calculates the height of a (not necessarily balanced) binary tree. A tree is built from a `treenode` data structure:

```
domain treenode {  
    domain treenode *parent, *leftson, *rightson;  
    int height;  
} root, othernodes[10000];
```

Here it is assumed that subtrees are not shared, so each node can have pointers not only to its sons but also a back-pointer to its (unique) parent node. The root node will have a null parent pointer, and leaf nodes will have null `leftson` and `rightson` pointers.

To compute the height of the root (in fact the height of *all* nodes):

```
[domain treenode].{
    int oldheight;
    height = 0;
    if (parent)
        do {
            oldheight = height;
            parent->height >?= height+1;
        } while (height != oldheight);
}
```

The height of a leaf node is to be zero; here we in fact initialize the heights of *all* nodes to be zero. Then all nodes that have no parent (that is, the root) drop out (because of the *if* statement), and the rest perform a loop. At each step, every node that has a parent takes its own height, adds one, and performs a maximum-assignment of this value to its parent's height. Of course, if the parent has another son, both sons are performing this maximum-assignment in parallel. The loop terminates when the heights stop changing, as determined by an OR-reduction, using the unary *!=* operator. At that point *root.height* will contain the height of the tree, and in fact every node will have its own height (where the height of a leaf is zero). Indeed, there is nothing in this code that particularly distinguishes the node named *root*; the same code can process a forest of trees, processing it in time proportional to the height of the highest tree. Even more interesting, the code makes no reference to the *leftson* and *rightson* components, only to *parent*, and so the code is suitable for processing any trees, not just binary ones.

#### 5.4 Sequencing and a Broken Symmetry

There is a subtlety in this code that must not pass without discussion. The As-If-Serial Rule makes it somewhat easier for a programmer accustomed to sequential thinking (as are we all) to reason about the behavior of a C\* program, but there are two points to be kept in mind. The first point is that the As-If-Serial Rule guarantees that *some* sequential order will be used, but it does not say *what* sequential order will be used. This might cause problems were it not for the second point, which is that the As-If-Serial Rule is to be applied on an operator-by-operator basis, not a statement-by-statement basis; in effect, the operands to each operator are fully evaluated for *every* processor before the operator itself is executed for *any* processor.

In the tree example we can see the importance of this second point. In the statement

```
parent->height >?= height+1;
```

one must realize that not only is a given processor updating its parent's *height* component, but one or more other processors (its sons) may be updating its *height* component at the same time. It is therefore crucial to the correctness of the algorithm that the expression

`height+1` be evaluated in all processors before any `height` component is altered by the `>?=` assignment operator.

Another consequence of this second point is an interesting breaking of symmetry. Every C programmer knows that using an assignment operator

```
x += y;
```

is equivalent to writing

```
x = x + y;
```

(except that the expression `x` is evaluated only once in the first case and twice in the second case). This is not quite correct in C\*. The equivalence holds in the single-processor case—including serial code, where we require C\* to match standard C semantics, as well as parallel code where only one data processor happens to be active—but fails when more than one processor is active. It is instructive to see why. Imagine many processors to be active. For the statement

```
x += y;
```

first all processors compute `x` (as an lvalue), then all processors compute `y` (as an rvalue), and finally all processors, as if in some serial order, each add the `y` value to the `x` location. The result is that every processor has added its `y` value into location `x`. But for the statement

```
x = x + y;
```

first `x` and `y` are computed as rvalues, and their sum (call it `s`) is then formed for every processor; also `x` is computed as an lvalue (either before or after `s` is computed; it doesn't matter for this example). The last step is that every processor then performs the assignment of `s` to location `x`, as if in some serial order. The net effect is that *some* processor (the last one in the effective serial order) will have added its `y` value to location `x`, and the `y` values of all other processors will have had no net effect on location `x`.

We can summarize this symmetry breaking as follows: the statement

```
x += y;
```

allows *every* active processor to add its `y` into `x`, whereas the statement

```
x = x + y;
```

allows *some* active processor (if any) to add its `y` into `x`. When there is exactly one active processor, these two statements amount to the same thing (to choose *some* processor is to choose *every* processor); but they do not mean the same thing when more than one processor is active.

### 5.5 Implementation on the Connection Machine System

C\* presents to the user a uniform treatment of pointers and assignment operators. It should not be surprising, however, that various special cases are translated by the compiler into different code sequences that are supported by different hardware resources.

Serial code that operates on mono data is treated as ordinary C code, and is executed on the front end. Serial code that accesses poly data in sequential fashion (*reading* and *writing*) is also executed by the front end, using the memory data bus to access the memory of the data processors.

*Replication* of mono values is done by broadcasting over the instruction bus, as described in section 3. *Reduction* of many values to one values, as for the expression `+=x`, is done with a combination of hardware features. First the router is used to treat the data processors as a binary tree. (This tree is organized using address arithmetic, not explicit pointers.) The data processors exchange and sum values in parallel, producing a result in time logarithmic in the number of processors. The front end then reads the sum from one particular processor through the memory bus. A statement of the form

```
mono-lvalue += poly-value;
```

which at face value seems to imply that data processors must be able to access the front end memory directly, is compiled as if it had been written

```
mono-lvalue += (+= poly-value);
```

which has the same net effect.

*Permutation*, *multiple broadcasting*, and *multiple reduction* is all done through the Connection Machine router, which provides not only message transmission but also support for broadcasting and reduction operations.

We can see, then, that a simple indirection expression `*p` or a compound assignment `x+=y` may be compiled in a variety of ways, depending on whether it appears in serial or parallel code and depending on whether the operands are mono or poly. However, the resulting code can be predicted from the data types involved, just as it can be predicted from operand data types whether a given `+` operator will result in an integer addition, a floating-point addition, or an indexing calculation.

### 5.6 Summary of C\* Expressions

- There are no special operators for parallelism, although maximum and minimum operators are added for their general utility, and unary assignment operators are used as a convenient abbreviation for a common idiom.
- mono values are replicated and broadcast when necessary.
- A statement of the form

```
mono-lvalue += poly-value;
```

performs sum-reduction; any of the C assignment operators may be used in this manner.

- A statement of the form `*p = x`; sends the message `x` to place `p`.
- An expression `*p`, where `p` is a poly pointer, can perform not just parallel interprocessor fetches, but in fact multiple broadcasts if there are several sets of pointers with each set pointing to the same place.
- A statement of the form `*p += x` can perform multiple reductions, one at each target location.

## 6 Parallel Statements

C\* adds only one new type of statement to C, the selection statement, which is used to activate multiple processors.

All of the standard C statement types may be used in C\* in both serial and parallel code. The treatment of control flow in parallel code satisfies the following design goals:

- As long as processors do not interact, the program behaves as if each processor were executing its own code independently. It is as if each of the parallel processes were executing ordinary serial C code.
- When processors do interact, the interactions are completely predictable, deterministic, and repeatable. This is achieved without ever requiring the programmer to write explicit synchronization code.

### 6.1 Selection Statement

A selection statement looks like this:

```
[domain tag] . statement
```

A selection statement activates all instances of a specified domain and then executes a substatement. (As with the `switch` statement, in theory the substatement may be any statement but in practice it is typically a block.) On completion of the substatement, the instances activated by the selection statement are deactivated.

Within the substatement, the keyword `this` is bound to the primal parallel value: for each active instance, `this` is a pointer to that very instance. Because writing the name of a member variable `memvar` is equivalent to writing `this->memvar`, all references to such a variable also constitute parallel values. (Note that uses of member variables that are qualified by non-parallel object pointers are still non-parallel.)

The selection statement is the means by which serial code can initiate parallel execution. The selection statement may also be used within parallel code; in this case all instances active just before execution of this statement become inactive, and on completion of the statement the same instances become active again.

(Previously active instances must first be deactivated for the following reason. Domain definitions have a hierarchical structure. Unrelated domains cannot share code, and cannot be simultaneously active, because unrelated domains have unrelated collections of member variables, laid out in unrelated ways. Code that accesses a member variable within one domain would produce nonsensical results within another domain. Thus, the set of all the instances of a given domain type is a maximal reasonable active set; that is what the selection statement activates. Subsets of this maximal set may then be selected by examining member variables and using if statements.)

## 6.2 If statement

In parallel code, the expression in an if statement is treated as a poly value, so that each active domain instance has its own value for the test. (If the expression is not poly, then one may regard the parallel if statement either as behaving like an ordinary serial if statement or as first casting the value of the expression to be poly, thereby replicating it. These two points of view are equivalent.)

For the statement

```
if ( expression ) statement
```

the *statement* is executed with only those instances active whose test value was non-zero.

**Rule of Local Support:** If every instance calculates zero for the expression then the statement is not executed at all. This must be stated explicitly, because it makes a difference if the *statement* modifies mono variables.

For the statement

```
if ( expression ) statement else statement
```

the first substatement is executed with only those instances active whose test value was non-zero, and then the second substatement is executed with only those instances active whose test value was zero. Again, if the set of active instances is empty for either substatement then that substatement is not executed at all.

The Rule of Local Support (that a substatement is not executed at all for an empty set of instances) occasionally causes some inconvenience, if not also confusion, when a parallel if statement is intended to conditionalize essentially serial code (code dealing with mono quantities). We have already noted that the statement

```
[domain book].{
    if (owner->salary < 40000.0)
        printf("%d books\n", +=(poly)1);
}
```

will not print anything if there are no book owners that earn less than 40000.0. To cause something to be printed in that case as well one must resort to some circumlocution such as this:

```

{
    int book_count = 0;
    [domain book].{
        if (owner->salary < 40000.0)
            book_count += (poly)1;
    }
    printf("%d books\n", book_count);
}

```

That is, the solution is to move the serial code back out of the selection statement into the serial world. Note that `book_count` must be explicitly initialized to zero.

(Because of these occasional difficulties of expression, it has been suggested that "serial" code occurring within a substatement of a parallel `if` should always be executed. Such a suggestion is based on an implementation model in which a parallel `if` statement is processed unconditionally from the point of view of the front end, which plows straight through the code for both arms of the `if`, enabling first one set of processors and then another set. This model is reasonable, and in the current implementation it is indeed some extra trouble to implement the Rule of Local Support. We have nevertheless adopted this rule for one very important reason: it is required for the goal that individual processors obey serial C semantics when not interacting. In particular, without this rule the usual model of a `while` loop in terms of `if` statements (shown in the next section) fails utterly. Maintaining such a useful general principle is far more important than minor conveniences of either programming or implementation.)

### 6.3 While statement

In parallel code as in serial code, the statement

```
while ( expression ) do statement
```

is completely equivalent to

```

if ( expression ) {
    statement
    while ( expression ) do statement;
}

```

or to

```

{
    Top:
        if ( expression ) goto Done;
        statement
        goto Top;
    Done: ;
}

```

(except that these models do not, as shown, reflect the action of `break` and `continue` accurately).

On each iteration the *expression* is calculated as for an `if` statement. Instances that calculate the value zero become inactive; instances that calculate a non-zero value execute the substatement and then loop. The set of active instances can change on each iteration only by decreasing monotonically as some instances drop out. (Instances can drop out either by calculating a zero value for the expression or by executing a `break`, `goto`, or `return` statement.)

The `while` loop completes if and when the active set becomes empty. At that time each individual processor has executed the substatement some number of times, and each may have executed it a different number of times, depending on the data being processed.

If the processors do not interact during the course of the loop, then it is as if each processor executes the `while` statement independently, each iterating the appropriate number of times, and then all processors become resynchronized when all have completed.

If the processors do interact, then their interactions are predictable; for example, all processors that execute the substatement as many as three times will all be executing it for the third time together, guaranteed.

#### 6.4 Goto statement

As described in the previous sections, the `if` and `while` statements behave in a reasonably intuitive manner in parallel code. This definition of their behavior, however, far from being *ad hoc*, is a consequence of a general (and admittedly rather complicated) model of control flow. The complexity of the model arises from the fact that it handles parallel `goto` statements in a general way, giving some of the characteristics of a MIMD execution model within the superficially SIMD framework of C\*. The full generality of the model is seldom used in practical C\* programs. The model is valuable for several distinct reasons:

- It explains the behavior of all types of statements in a uniform way.
- It produces the behavior for such statement types as `if` and `while` that one might expect either intuitively or from examination of the standard models for these statements in terms of `goto`.
- When processors are not interacting, each processor behaves exactly as if it were executing ordinary sequential C code.
- Processor interactions are synchronous and predictable from the model.

Readers who prefer not to dive into the details of the C\* control flow model may wish to skip to section 7.

Serial code in C\* is executed as ordinary C code in the usual serial manner. In this mode there is a single "locus of control" (a single "program counter," or PC, if you will) that moves through the program text, jumping around on occasion under the direction of

control statements and function-call operations. Code is executed as the locus of control moves past it.

Parallel code is handled internally in a rather different way. The overall idea is that such parallel code should behave as if each processor were executing a separate copy of the code in a serial fashion, while nevertheless remaining synchronized with the other processors. In the parallel mode there is still a single locus of control—we will call this the “master PC”—and code can be executed only as the master PC passes through it. However, there is also a “latent PC” for every processor in the processor array belonging to the currently selected domain. A latent PC may be “active,” in which case it is at the same place as the master PC and is moving along with it, or it may be “waiting,” in which case it is at some other point in the code. A processor whose latent PC is active participates in the parallel execution of expressions of poly type; a processor whose latent PC is waiting does not participate.

**Rule of Merging Control:** If the master PC, in the course of execution, passes a point where one or more than one latent PC is waiting, then every such latent PC becomes active and proceeds from that point with the master PC.

**Block Synchronization Rule:** Whenever the master PC reaches the end of a statement, every active latent PC becomes inactive and waits at a point just after the statement. Execution then ceases, and the master PC is immediately transferred to a new point in the code. (Call the point at which it was stopped the “old point.”) The new point is chosen as follows: one examines the statement containing the old point, then the one containing that, and so on, until one finds a statement that has at least one latent PC waiting within it; the new point is the textually earliest point within that statement that has a latent PC waiting. (In practice this is frequently the same as the old point, if there have been no complicated transfers of control, and the C\* compiler can usually determine the new point at compile time.) This is called the Block Synchronization Rule because one important consequence is that once the master PC enters a block, it does not leave the block until every latent PC has left the block.

The following additional rules are needed for handling specific statement types. There is nothing surprising in them; they simply restate the normal semantics of each statement type using the terminology of the C\* control flow model.

When a selection statement activates parallel processing, one latent PC is created for every instance of the selected domain, and these start out active with the master PC at the head of the body of the selection statement. When the body is finished, the every latent PC created for that execution of the selection statement is destroyed.

When the control expression of an `if` statement that has an `else` part is executed, and that expression is parallel, then the latent PC for each processor that computed a zero value is moved to just after the `else` and waits.

When the control expression of a `while`, `do`, or `for` statement, or an `if` statement with no `else`, is executed, and that expression is parallel, then the latent PC for each processor that computed a zero value is moved to just before the end of that statement and waits.

When the master PC reaches an `else`, then every active latent PC is moved to just before the end of the `if` statement and waits. When the master PC reaches a `switch` statement, then every active latent PC is moved to just after some `case` label (corresponding to the value it computed for the `switch` expression), and waits. When the master PC reaches a `break`, `continue`, or `goto` statement, then every active latent PC is moved to the indicated point in the code and waits (but the master PC remains behind, exits the `break`, `continue`, or `goto` statement, and is then transferred to a new point in the code according to the Block Synchronization Rule).

These rules may seem strange, but this is only because they cover some rather strange general cases. They do cause important special cases to behave in the ways that might be expected. For example, when an `if` statement is encountered, first those processors that compute a nonzero value execute the first substatement (the “then” part), then all other processors execute the first substatement (the “else” part), and finally the two sets of processors are rejoined for continued execution. The rules also cause loops to behave in the expected way, even if the user synthesizes them out of `if` and `goto` instead of using the built-in `while`, `do`, and `for` statements. Finally, statement structure (including block structure) is respected; if several different latent control points get to flowing around within a statement, none is allowed to continue execution outside the statement until all have left the statement. This guarantees that all the processors that leave a block by falling off the end (rather than by an explicit transfer of control) will become resynchronized at that point.

This model allows `goto` to be used in a general way, and specifies precisely when and how processors will become resynchronized. This is not to say that it is easy to reason about the behavior of `goto`-ridden code in the general case.

### 6.5 Continue, Break, and Return Statements

The standard serial model of `continue` in terms of `goto` also applies in the parallel case. To elaborate: in parallel code the `continue` statement causes all active instances to transfer to the loop-continuation portion of the smallest enclosing `while`, `do`, or `for` statement. There they wait to be reactivated. They may continue execution immediately, or may have to wait while other instances still within the body of the loop continue execution.

The `break` and `return` statements are handled in a quite similar manner, and are not discussed further here.

### 6.6 Switch statement

A `switch` statement in C may be understood as a multiway `goto` statement, and this understanding carries over directly to the C\* model. In parallel code, the `switch` statement causes every active instance to perform something like a `goto` so that control for that instance is logically transferred to a `case` label within the body of the `switch` statement. Execution then continues from the textually earliest such label, activating only instances that had transferred to that label. If execution subsequently passes through a `case` label,

any inactive instances waiting there become active and join in the execution. If a **break** statement is executed, all active instances become inactive and wait at the end of the **switch** statement, and if any previously inactive instances are still waiting at a **case** label within the same **switch** statement then execution continues from the earliest such label. By this means all instances are advanced through the body of the **switch** statement.

If and when all instances (other than those that execute a **continue**, **return**, or **goto** statement that transfers out of the **switch**) have reached the end of the **switch** statement, they all then proceed together to whatever follows the **switch** statement.

Example: suppose that five domain instances *a*, *b*, *c*, *d*, and *e* are active, and that they have the following values, respectively, for the poly variables *x* and *y*:

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>
<i>x</i>	6	6	7	9	7
<i>y</i>	0	1	0	0	1

Also assume that *p* and *q* are mono variables whose values are initially zero.

Consider then this **switch** statement:

```
switch (x) {
  case 5:
    x = 43;
  case 6:
    p += (poly int) 1;
    if (y) break;
    x = 13;
  case 7:
    q += (poly int) 1;
    x = 27;
    break;
  case 8:
    x = 6;
  case 9:
    x = 91;
}
```

Execution proceeds as follows. The **switch** causes instances *a* and *b* to wait at label case 6, *c* and *e* to wait at label case 7, and *d* to wait at label case 9. Because case 6 is the textually earliest point in the statement with waiting processors, execution picks up from there with instances *a* and *b*. The assignment to *p* is performed; because two instances are active, *p* is incremented by two. Next *y* is tested. Instance *b* has a non-zero value, so it executes the **break** substatement and waits at the end of the **switch** statement. Instance *a* proceeds to assign 13 to *x*.

Execution proceeds with instance *a* and drops through the label case 7, where instances *c* and *e* are waiting. All three instances then execute the assignment to *q*, thereby

## §7 A Large Example

incrementing it by three, and then assign 27 to `x`. All three instances then unconditionally execute the `break` statement and wait at the end.

Instance `d` is still waiting at label `case 9`, so execution picks up from there. Instance `d` assigns 91 to `x` and then drops off the end of the `switch` statement. There are no other instances waiting earlier in the `switch` statement, so all five instances proceed together to whatever follows the `switch` statement.

On completion of the `switch` statement by all instances, the variable `x` has been altered, so:

	<code>a</code>	<code>b</code>	<code>c</code>	<code>d</code>	<code>e</code>
<code>x</code>	27	6	27	91	27

Note that instance `a` assigned to `x` twice, first 13 and then 27. The variable `p` has the value 2 and `q` has the value 3.

## 7 A Large Example

As an example of a moderately substantial chunk of C\* code, we present here a small piece of an  $N$ -body simulation: given the positions and masses of  $N$  particles, to compute for each particle the net instantaneous acceleration induced gravitationally by the other particles. The code implements the obvious brute-force  $O(N^2)$  algorithm. The point of this example is not the algorithm *per se*, but the ease with which the algorithm can be expressed.

We will need two kinds of data items, one to represent particles and one to represent particle interactions. For a particle we require `x`, `y`, and `z` coordinates for position, mass, and places in which to store the resulting `x`, `y`, and `z` acceleration components. An interaction instance needs to identify the two particles with which it is associated.

```
domain particle {  
    float x, y, z, mass, ax, ay, az;  
};
```

```
domain interaction { int j, k; };
```

For  $N$  particles we will need  $N^2$  interactions. (Of these  $N^2$  interactions,  $N$  will be wasted, for a particle has no effect on itself. We could have allocated an  $N \times (N - 1)$  array instead of an  $N \times N$  array; this would have required complicated index expressions in the code. For the sake of the example we avoid such complications.)

```
domain particle P[N];  
domain interaction I[N][N];
```

The following function initializes all the interaction instances so that each has a different `(j, k)` pair. Note the use of address arithmetic on the variable `this` (whose value will be different within each instance) to compute the distance of each instance from the origin of the array `I`.

```

void interaction::setup() {
    int offset = (this - &X[0][0]);
    j = offset / N;
    k = offset % N;
}

```

The code to compute the accelerations follows.

```

void compute_accelerations() {
    [domain particle].{ ax = ay = az = 0.0; }
    [domain interaction].{
        /* Compute the acceleration induced
         by P[j] on P[k]; each interaction point
         has a different (j, k) pair. */
        if (j != k) { /* No point acts on itself. */
            float dx = P[j].x - P[k].x,
                  dy = P[j].y - P[k].y,
                  dz = P[j].z - P[k].z;
            float r_squared = dx*dx + dy*dy + dz*dz;
            float r_cubed
                = r_squared * sqrt(r_squared);
            float q = G * P[j].mass / r_cubed;
            /* All the updates happen at once! */
            P[k].ax += q*dx;
            P[k].ay += q*dy;
            P[k].az += q*dz;
        }
    }
}

```

First every particle is activated in order to clear the acceleration components *ax*, *ay*, and *az*. Then all  $N^2$  interaction instances are activated. Interactions for which *j* equals *k* are then disabled by the *if* statement, leaving  $N^2 - N$  processors active. Each of these processors then computes the relative distance in each coordinate (*dx*, *dy*, and *dz*) between particle *k* and particle *j*. The square and cube of the straight-line distance is then computed for each pair, and then an expression *q* that is common to all the acceleration components. (Here *G* is assumed to be globally defined as the gravitational constant.) In each of the last three assignment statements one acceleration component is computed and then added into the appropriate member of particle *j*. In each such statement note that the  $N^2 - N$  active processors may be divided into *N* groups corresponding to the *N* particles; each group contains  $N - 1$  processors that have computed the acceleration components resulting for that group's particle from the force exerted by the other  $N - 1$  particles. Execution of the assignment operator *+=* therefore represents the performing of *N* sum-reductions, each of  $N - 1$  terms.

## 8 Compiler Implementation

The C\* compiler for the Connection Machine computer system is implemented as a translator to ordinary C code that is then compiled by an ordinary C compiler for the front end computer. The C\* compiler parses the C\* source code, performs type and data flow analyses, and then translates parallel code into a series of function calls that invoke PARIS operations (PARIS is the Connection Machine PARallel Instruction Set [12]).

The use of the front end's usual C compiler allows all the programming tools associated with the front end programming environment to be applied to C\* programs; of course, some of these tools (the debugger in particular) have been extended for convenient handling of parallel values.

## 9 Comparisons with Other Work

Vector C [10] extends C by allowing arrays in effect to be treated as first-class objects (vectors) by using a special subscripting syntax to select array slices. The new syntax is quite versatile, allowing periodic scatter/gather operations, compress/expand operations, and vector-valued subscripts. A new data type, the vector descriptor, is also introduced. The standard C operators act elementwise on vectors, and some twenty new expression operators (each beginning with “@”) provide additional vector operations. For example, if  $x[*]$  and  $y[*]$  are vectors then  $x+y$  is the elementwise sum of  $x$  and  $y$ ,  $@x$  is the sum of all the elements of  $x$ ,  $@>x$  is the largest element of  $x$ , and  $x@.y$  is the vector inner product of  $x$  and  $y$ .

Vector C allows convenient manipulation of vector values. However, we believe that it does so by substantially changing the feel of the C language. It introduces a large number of new (and useful) operators that have no analogue in ordinary C. Moreover, Vector C relies on a view of arrays as first-class objects, whereas the confusion of arrays with pointers is essential to the flavor of the C language. C\* differs from Vector C in providing fewer special facilities for vector manipulation (vector inner product is not a primitive in C\*, for example, though it can be expressed as  $+=(x*y)$ ), in preserving the interchangeability of arrays and pointers, and in relying more on a local view of computation on elements than on a global view of operations on arrays.

Refined C [1] introduces some interesting notation for partitioning arrays. However, the language design is oriented not toward the expression of data level parallelism, but rather toward aiding a compiler, through various declaration and type mechanisms, to extract control level parallelism from a C program. It is noteworthy that Refined C, like C\*, adopts the C++ class notation in order to organize data, but for a different purpose. The ways in which Refined C extends C are claimed to be fairly independent of the particulars of the C language; reference [1] also exhibits a sample program in Refined Pascal, and reference [2] describes a similar extension of Fortran.

PASM Parallel-C [9] is perhaps closest in spirit to the design of C\*. Parallel-C allows *any* data type to be declared in a parallel form, whereas in C\* the domain is the only parallel data type (all other parallel quantities being members of domains). In

some ways Parallel-C treats parallel data types as first-class arrays, in the manner of Vector C, but Parallel-C does not introduce so many new operators. Parallel C can select a subset of array elements either through special index expressions (especially those involving a new data type called a *selector*) or by use of an *if* statement with a parallel test expression. C\*, of course, uses only the latter mechanism. Reference [9] enumerates the four cases for assignment statements (scalar/scalar, scalar/parallel, parallel/scalar, parallel/parallel) and discussed each separately. The scalar/scalar and parallel/scalar cases are treated exactly as in C\*, but in Parallel-C the other two cases are greatly restricted. If the left-hand side is scalar and the right-hand side is parallel, then the parallel side must have exactly one element selected for the result of the assignment to be defined. Compound assignment operators in Parallel-C are not discussed. It is of course exactly this case that gives C\* much of its expressive power, by allowing assignment operators to perform reduction. If both sides are parallel, Parallel-C performs assignment only for corresponding elements, and again reduction is not possible; Parallel-C requires the use of special functions for interprocessor communication. Parallel-C allows processors to operate in either "SIMD mode" or "MIMD mode"; reference [9] does not describe the MIMD mode features in great detail, but does indicate that the programmer is responsible for using explicit synchronization to maintain data integrity.

C\* is apparently the first parallel C language of its style (no explicit synchronization constructs) to allow the use of all standard C control statements, including *goto*.

## 10 Future Work

One disadvantage of the current C\* control flow model is that conditional statements are apparently wasteful of resources, or at least of potential parallelism: while one branch of a conditional is being executed, the other is definitely not, even though they would be processing disjoint sets of data elements and the processing might not interact. It may be possible to "loosen up" the control flow model in limited ways to allow exploitation of this kind of control parallelism to complement data parallelism. The challenge is to achieve this without losing all the advantages of deterministic, predictable, repeatable execution that are provided by the completely synchronous model. We believe these advantages should not be underestimated, because they can significantly affect total productivity of both human and computer resources.

At present not all aspects of C++ have been integrated into C\*; for reasons of simplicity and expedience, only those features necessary to support data parallel programming have been adopted. Features not adopted include information hiding through non-public members, class derivation, and user overloading of built-in operators. However, C\* has carefully been designed to allow later introduction of these features to produce a language with the parallel capabilities of C\* as well as all the object-oriented and information-hiding features of C++. (The obvious name for the resulting hybrid language is of course \*C++.)

## 11 Conclusions

C\* allows the writing of parallel programs while building on familiar paradigms from C and C++. Algorithms are coded in a superficially SIMD style, but MIMD like control flow semantics are provided. There is no need for the programmer ever to express process synchronization explicitly. To the extent that processors do not interact, each behaves as if executing ordinary sequential C code. When they do interact, their behavior is always deterministic, predictable, and repeatable.

C\* was designed to support the writing of programs in a data parallel style. It is therefore suitable for programming computers designed to support data parallel processing. However, because of the simplicity of the C\* machine model, and its lack of dependence on specific architectural features, C\* can also be used to program other kinds of computers. We believe that C\* provides and promotes a style of programming that is useful even on sequential computers. As long as the machine in question can efficiently apply the same operation to a set of many operands, then it can execute C\* programs efficiently, maintaining the illusion of many processors at the language level.

## 12 Acknowledgements

The C\* language was designed by John Rose, Guy Steele, and Stephen Wolfram.

The C\* compiler for the Connection Machine computer system was programmed by John Rose, with assistance from Sam Kendall, Guy Steele, Skef Wholey, Marshall Isman, and J. P. Massar.

Jim Bailey and Ted Tabloski made substantial contributions to improving the presentation of this paper. Keira Bromberg and Gary Rancourt were also helpful in preparing the text. Arlene Chung and Kate Reichart prepared some of the illustrations.

## References

- [1] Dietz, Henry, and Klappholz, David. Refined C: a sequential language for parallel programming. In Degroot, Douglas, editor, *Proc. 1985 International Conference on Parallel Processing*. IEEE Computer Society (August 1985), 442-449.
- [2] Dietz, Henry, and Klappholz, David. Refined Fortran: another sequential language for parallel programming. In Hwang, Kai, Jacobs, Steven M., and Swartzlander, Earl E., editors, *Proc. 1986 International Conference on Parallel Processing*. IEEE Computer Society (August 1986), 184-191.
- [3] Gilman, Leonard, and Rose, Allen J. *APL: An Interactive Approach*, second edition. Wiley (New York, 1976).
- [4] Harbison, Samuel P., and Steele, Jr., Guy L. *C: A Reference Manual*, second edition. Prentice-Hall (Englewood Cliffs, New Jersey, 1986).

- [5] Hillis, W. Daniel. *The Connection Machine*. MIT Press (Cambridge, Massachusetts, 1985).
- [6] *APL\360 User's Manual*. International Business Machines Corporation (August 1968).
- [7] Iverson, Kenneth E. *A Programming Language*. Wiley (New York, 1962).
- [8] Kernighan, Brian W., and Ritchie, Dennis. *The C Programming Language*. Prentice-Hall (Englewood Cliffs, New Jersey, 1978).
- [9] Kuehn, James T., and Siegel, Howard Jay. Extensions to the C programming language for SIMD/MIMD parallelism. In Degroot, Douglas, editor, *Proc. 1985 International Conference on Parallel Processing*. IEEE Computer Society (August 1985), 232-235.
- [10] Li, Kuo-Cheng. A note on the Vector C language. *ACM SIGPLAN Notices* 21, 1 (January 1986).
- [11] Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley (Reading, Massachusetts, 1986).
- [12] *Connection Machine Parallel Instruction Set (PARIS), Release 2, Revision 7*. Thinking Machines Corporation (Cambridge, Massachusetts, July 1986).
- [13] Wagener, Jerrold L. Status of work toward revision of programming language Fortran. *ACM FORTEC Forum* 3, 2 (May 1984), 1-42.

1. The first step is to identify the problem.

2. The second step is to analyze the problem.

3. The third step is to design a solution.

4. The fourth step is to implement the solution.

5. The fifth step is to test the solution.

6. The sixth step is to document the solution.

7. The seventh step is to maintain the solution.

8. The eighth step is to evaluate the solution.

9. The ninth step is to report the results.

# Possible Directions for C++

Bjarne Stroustrup

## ABSTRACT

C++ has shown itself an effective tool for programming in an extremely wide range of application areas. This paper examines the strengths and weaknesses of C++ and its associated tools in order to see what developments might yield significant further advantages to C++ users. The main topics for discussion are libraries, container classes, parameterized types, exception handling, incremental compilation, incremental linking and loading, object I/O, persistent objects, automatic storage management, and design methods.

## 1 Introduction

A programming language is only a small part of a programmer's world, but a very important part. The tools, techniques, libraries, and culture that surround a language, constituting what is often referred to as the programming environment, are of crucial importance.

No major program is ever written in the programming language as described in its basic language manual; libraries of all sorts are used and often determine the structure of the program. The naive user cannot even distinguish between a language and its basic support libraries; the experienced user often attaches so much importance to the available libraries that the language itself is considered of secondary importance.

Similarly, the compilation (or interpretation) environment determines the approach to program development, debugging, and testing. This again can shift the focus of attention away from the basic language.

However, a language is typically a crucial factor in the design of tools such as compilers and debuggers and in the design of libraries. This paper focuses on C++ as a base for building and using libraries and as a raw material for designers of programming support tools. The programming environment is considered from the point of view of how it can support such library building and how it can support the styles of programming that occur when large numbers of high-quality libraries are available.

My notion of high quality involves the idea of a definite structure of a program that can be documented and the idea that a library must be affordable. In C++, the structure is supported by a strong static type system with a flexible and general notion of encapsulation and interfaces. The concern for affordability becomes an efficiency requirement: it doesn't matter how elegant a library is if it is too expensive to use.

C++ is taken to be the language as exists today (October 1987) with multiple inheritance, recursive memberwise assignment semantics, protected members, pointers to members, etc. For details of the evolution of C++ from the publication of *The C++ Programming Language*<sup>10</sup> to now see reference 12.

Naturally, the areas for improvement of C++ and its related tools and techniques considered here do not constitute an exhaustive list of areas in which one might do useful C++ related work. For example, the absence of a discussion of debuggers does not imply that nobody needs a C++ debugger. Furthermore, a mention of a solution or a technique here does not imply that that technique is the only solution to the problem it was designed to solve.

## 2 Libraries

The standard libraries shipped with a C++ translator are:

- streams for uniform, flexible, type-safe I/O;
- tasks for simulation and other applications requiring light-weight processes;
- complex numbers and basic complex functions;
- a set of macros for "faking" parameterized types;
- parameterized vector and stack classes.

In addition, many specialized libraries are available in local environments. Several obvious candidates for the standard library are in wide use within AT&T. For example:

- strings with a full set of operations;
- maps; that is, associative arrays;
- lists of various sorts;
- counted Pointers for automatic storage management of specific objects.

Most C++ users simply use the standard and not so standard C and Fortran libraries found on UNIX, VMS, MS-DOS, OS/2, or whatever operating system they use. This is a very important source of libraries since it grants the C++ programmer access to millions of lines of tested code.

The basic strength of C++ for library building is its facilities for defining efficient and type-safe classes where a single underlying type exists for a problem area. Examples are characters for string manipulation and pattern matching, floating point numbers for many kinds of numerical work, and bits, bytes, and pointers for many kinds of systems work. The basic weakness of C++ for library building is its lack of special features to support classes for which no such single underlying type exists. The best examples of such classes are container classes, such as sets, vectors, lists, and associative arrays.

### Container Classes

The standard libraries that come with languages such as Smalltalk<sup>2</sup> and the object-oriented Lisp dialects<sup>1,13</sup> show one possible direction of development. This style, relying mostly or even exclusively on dynamic type checking, can be used in C++ as demonstrated by Keith Gorlen's OOPS library<sup>4</sup>. This leads to flexible and general libraries suitable for open-ended program development. Such libraries have proven useful for exploratory and prototype development but high run-time cost has made such libraries uneconomical in applications with heavy computational components. There have also been problems in ensuring correctness of programs in large multi-person projects due to the lack of statically specified and verifiable interfaces.

An alternative style of library building can be found in the "generic" or "parameterized" types in Clu<sup>6</sup> and Ada<sup>3</sup>. Here, strong static typing is maintained with the obvious result of increasing efficiency while providing a more definite and solid program structure. To achieve run-time efficiency comparable with that of built-in types the basic operations on instances of such parameterized classes must (on most machine architectures) be implemented without function call overhead. Jonathan Shapiro's list classes<sup>7</sup> and Andy Koenig's associative array classes<sup>5</sup> demonstrate that this style can be used in C++. However, this style of libraries is not as flexible as the Smalltalk and Lisp libraries and not as supportive of the open-ended style of programming as one might like.

The Smalltalk-style of containers has the important benefit of providing a single common concept of a container from which all specific containers are derived. This allows a user to write code that does not have to deal with details of any particular container. For example, it is trivial to write a function that accepts any kind of container.

This is harder to achieve in the Clu-style where there is not necessarily a single concept of a container, just individual containers such as sets and sorted lists. The advantage of this approach is that the overhead of operations can be minimal; less than a function call for the simplest operations. Using the Smalltalk-style makes every operation on a container a method invocation. In C, of course, one would use special "hand coded" containers

with minimal overhead. For the standard containers to be genuinely useful it must be possible to approximate the efficiency (in time and space) of the C approach where it matters.

The ability to write functions that depend only on a general concept of a container rather than on properties on individual containers is fundamental to the design of general utility functions and classes. If the Clu-style is adopted it must be augmented by a unifying container class concept.

A very attractive solution is to combine the two approaches by utilizing a set of basic and standard parameterized classes to provide efficiency and a definite static structure. These parameterized classes can then be instantiated for base types to provide the code sharing and polymorphism necessary for large open-ended systems. C++'s virtual functions are ideal for this in that they provide a way for a base class to present a strongly typed interface to a variety of derived classes.

The key feature needed in C++ to support this style of programming is parameterized types. Currently, in C++ one must rely on macros to "fake" parameterized types, but that approach does not work well except on a small scale<sup>†</sup>.

### Parameterized Types

Assuming that C++ did have a proper set of features supporting parameterized types we could write:

```
#include <Vector.h> // make the standard vector class available
#include <Shape.h> // make the standard geometric shape class available

main()
{
    Vector<int> ivec(200); // vector of 200 integers
    Vector<Shape*> svec(10); // vector of 10 Shape pointers

    svec[0] = new Rectangle( ... );
    svec[1] = new Circle( ... );
    ...
    do_something(svec);
    ...
}

void do_something(Vector<Shape*> &v)
{
    for (int i = 0; i < v.size(); i++) {
        ...
        v[i]->draw(); // draw v[i] according to its type
        ...
    }
}
```

The Vector class might be declared like this:

<sup>†</sup> Use of macros interposes a level of translation between the programmer and the programming support tools. This often renders such tools ineffective, as when the compiler reports errors at the point of use rather than at the point of declaration for a macro. It also renders communication between different tools cumbersome, as when identifiers are unavailable to a symbolic debugger because they were expanded by the macro processor. Furthermore, the C preprocessor, `cpp`, used with C++ isn't elegant when it comes to multi-line macros and using it to "fake" parameterized types often presses both it and other tools to the limit. For example, it is not uncommon to find tools that cannot handle the 3000 character output lines you occasionally get from `cpp` when using these techniques.

```

class Vector<class T> {
    T* v;
    int sz;
public:
    Vector(int);
    ~Vector() { delete v; }

    T& operator[](int i);
    int size() { return sz; }
    ...
};

```

Why bother with container classes such as Vector? After all, if I want a vector of shapes I can write one in C++ in a couple of minutes. Here are some reasons for using a parameterized class rather than hand crafted classes:

- The more interesting container classes take much more than a couple of minutes to write.

- It is tedious and error-prone to re-write such classes for each element type.

- Programmers will soon learn to recognize and use standard container classes

- hence programs will become more readable (to the experienced programmer if not to the complete novice).

- Standard container classes are more likely to be correct and more likely to have been optimized in some way

- hence *both* program development and run-time performance will improve.

These benefits will only manifest themselves provided the use of such classes is both convenient and efficient enough. The macro approach is efficient enough, but not convenient. The Smalltalk-like use of dynamic typing to define standard container classes is convenient, but isn't efficient enough for many applications. Neither approach provides a proper type-safe foundation for the design of large systems. A proper mechanism for parameterized types is needed for that. Such a mechanism can and must be understood, relied on, and exploited by all major tools in the programming environment.

Type parameterization is most crucial for container classes because such classes can provide the glue used to bind systems together, but there are many other uses; probably the most obvious and important is that type parameterization will allow arithmetic functions to be parameterized over their basic number type so that programmers can (finally) get a uniform way of dealing with integers, single precision floating point numbers, double precision floating point numbers, etc.

## Exception Handling

In any language it is difficult to cope with errors that are detected in one context but cannot be properly understood and handled except in some other context. For example, a compiler often detects a type or syntax error while analyzing an expression but can only recover by returning to the routines handling the statement level. Similarly, a system output routine may encounter an unusual state of its device and need to return to a higher level routine that has sufficient information to decide what to do about it. In both cases a normal return through the sequence of calls that led to the point of error is tedious and error prone since the "normal state" of the computation cannot be assumed and there is typically no good value to pass back as the return value. Coping with this for a particular kind of "error" is tedious, but feasible. Coping with all such cases leads to an explosion in code size and complexity. Not coping with such cases leads to disasters.

In higher level languages, some form of "exception handling" mechanism is often provided as part of the programming language. In C, one usually relies on `setjmp()` and `longjmp()` to cope with the nastier examples of such errors. Because of destructors the C solution is insufficient and C++ will eventually be forced to develop an exception handling mechanism. Consider a class X for which a destructor is defined:

```

void f()
    X a;
    g();
    // here we know how to handle disasters
}

g()
{
    X b;
    h();
}

h()
{
    X c;
    // here a disaster happens
}

```

The problem is that if `f()` knows how to recover from the problem found in `h()` how does the control get back into `f()` in such a way that the destructors for `c` and `b` are called correctly?<sup>†</sup>

Here is what a solution might look like:

```

void f()
    X a;
    try { // exception handler defined for statements in this
        // scope and all statements reached from calls made
        // in this scope
        g();
    }
    except {
        // here we know how to handle exceptions
        // we only get here if an exception was raised
    }
}

g()
{
    X b;
    h();
}

h()
{
    X c;
    if (something_strange_happened) except! // raise exception
}

```

When the exception is raised the stack is unraveled and destructors called until a scope is reached in which an exception handler is defined. In this case, `X::~X()` is called for `c` and `b` before the exception handling code in `f()` is called.

There are many issues to consider when designing a mechanism for doing this in the context of C++. For example:

Can there be many different exceptions?

If so, how are they named?

<sup>†</sup> If they needn't be called the `setjmp()` approach is sufficient and the case is uninteresting. Getting `c` destroyed properly can be handled easily by the compiler; the difficult case is `b`.

What is a good syntax?

However, some of the more important design issues are settled by the nature of C++:

It is not possible for an exception handler to request that the program continue at the point where the exception was raised (destructors may have been called so that the context has been destroyed).

The exception handling mechanism must function properly even when C++ functions call non-C++ functions and when non-C++ functions call C++ functions.

The major run-time cost of the exception handling mechanism should be incurred when an exception is raised (and *not* at every function call or at every point where an exception handler is established).

The implementation should be very simple to port (or else the exception handling mechanism might become a bottleneck for increased C++ use).

The alternative to having such an exception handling mechanism is to avoid the use of classes with destructors in contexts where "exceptions" may occur and use `longjmp()` to handle the exceptions. This approach isn't feasible in an environment where standard libraries are heavily used because a user typically will not know whether destructors are used in a particular call chain.

An exception handling mechanism would also be used as the standard mechanism by which library routines signal errors to their users. The (current) alternative is to rely on a variety of ad hoc methods†.

### Special-Purpose Languages

Special-purpose languages can be an important source for ideas for library designers. Over the years, many languages have been designed to cope with specific problems either as special-purpose languages or as languages with "unusual" facilities supporting a specific application area. For example, consider the following languages as special-purpose languages (deliberately ignoring their general-purpose features):

Fortran	arithmetic
Snobol	pattern matching
APL	vector manipulation
Simula	simulation
Smalltalk	interactive user interfaces
Lisp	list manipulation
Awk	ASCII file manipulation
Prolog	logic programming
Ada	concurrency

The scope for "borrowing" features from such languages into C++ in the form of libraries rather than in the form of language extensions is large. For example, the simulation facility from Simula was relatively cleanly incorporated into C++ in the form of the task library and that library has recently been extended to cope with real-time for robot control<sup>8</sup>. The key here is C++'s ability to link to non-C++ code. In case of the task library, a few assembler instructions are used to manipulate machine registers to accomplish the basic co-routine context switch that cannot be implemented in C++ proper. For more delicate surgery, say for implementing a vector type using special vector operations available in a vector processor, asms in inline functions can be used to achieve non-C++ operations without the overhead of a function call.

The alternative to borrowing key ideas as C++ libraries is either to make special-purpose language extensions or to use a special-purpose language instead of C++. The problem with the first alternative is language size and complexity; the problem with the second

† It could be argued that an exception handling mechanism is therefore essential for the design of large high-quality libraries, but I don't want to push that argument too far.

alternative is that it requires communication between the special-purpose language and a general-purpose language to implement a complete system. Such communication is notoriously hard to achieve except where one can afford to go through some language independent medium such as a UNIX pipe. Where it is easy to achieve such communication, as with closely related languages such as C++ and C, there isn't a problem in the first place.

Library building is no panacea, but to avoid language fragmentation one should consider the library building approach very seriously before undertaking a project to extend the language by modifying a compiler or by providing yet another preprocessor. The likelihood is that any such extension will drastically reduce its users' ability to take advantage of alternative C++ compilers and/or other tools built in environments where the extension isn't in use. Furthermore, any two independently designed extensions are unlikely to be compatible or to possess facilities for convenient and type-safe interaction between the two extensions.

### 3 Environment Support

An environment can support the programmer in two ways: by supporting the activity of programming (providing tools for designing, specifying, implementing, and debugging programs) and by providing a congenial environment for the final program to execute in (standard utility libraries, operating system interfaces, etc.).

For C++ at least, there will always be several different development and execution environments and there will be radical differences between such environments. It would be unrealistic to expect a common execution environment for, say, an Intel 80286 and a Cray XMP, and equally unrealistic to expect a common program development environment for an individual researcher and for a team of 200 programmers engaged in large scale development. It is also clear, however, that many techniques can be used to enhance both kinds of environments and that one must strive to exploit commonality wherever it makes sense. For example, incremental linking of classes can be used both to decrease the time needed for the debug cycle during program development and to enable incremental build-up of a user-interface in an interactive application. The system-dependent aspects of incremental linking can also be localized so that a common style of incremental linking of classes can be available on a large range of systems.

Assume C++ were extended with parameterized types and exception handling. This would make design of libraries easier. What support could be provided in the programming environment to make programming more convenient? Consider the steps in a traditional C or C++ compilation.

[1] Run the preprocessor to incorporate the header files into the source.

[2] Compile.

[3] Link and search libraries for pre-compiled standard functions and data.

For C, the major part of the text processed (say 90%) is in the .c file that is often thought of as "the program." In C++, the header files are often larger than the .c file, and with a systematic use of libraries most of the source code (say 80%) will be in the header files defining classes, etc. Furthermore, in C most of the code in a final program typically comes from the user's (or users') .c files and relatively little (say 20%) is added from the library search. The libraries searched are typically small. This too will change. I expect most small C++ programs will search large libraries and that most (say 80%) of the final program will be found in some library or other.

The effect of all this is that whereas with C the speed of the compiler proper when processing .c files is the dominant factor in a compilation, with C++ processing headers, linking, and library lookup will dominate.

## Incremental Compilation

An important observation about the scenario outlined above is that between successive compilations a user only changes a small fraction of the files. Typically only a minor change to one or more .c files is done. This implies that if the cost of re-compilation can be made proportional to the size of the change, rather than the traditional "proportional to the size of the files in which the change occurred plus the size of header files included by those files", then compilation of C++ programs using large libraries would become faster than traditional C compilations. A back of the envelope calculation suggests that a 20 times improvement in apparent compiler speed is feasible. I conclude that we need incremental compilation and some form of incremental loading to complement it†.

Incremental compilation clearly requires that "the compiler" can determine what the increment is. In other words, there must exist a program that determines what has changed between two compilations and how much must be compiled because of this change (a C++ dependency analyzer). There are several ways of ensuring the correct use of such a set of compilation tools. Using an editor that understands both is a common technique, but there is nothing in the concept of incremental compilation that requires an integrated environment with a dedicated editor/browser, debugger, etc.

The C equivalent to this is simply cc and make. The problem with make for C++ programs (as it is for C programs) is that make doesn't understand C, let alone C++. Therefore make decides on what to re-compile based only on the file structure used to store the program and not the structure of the program itself. These decisions are typically far too conservative; that is, make often decides to re-compile files that are unaffected by a change.

One problem that will have to be looked into is that like C, C++ does not have a very strong notion of a "module" that helps a "smart" C++ environment to maintain a correspondence between program fragments and the way they are stored.

One extreme approach to this problem is to maintain the C convention of .c and .h files with no constraints on what kind of information is stored where. The opposite approach is to keep a program in a "program data base" of more or less compiled fragments. The storage of such fragments is organized exclusively by the program structure and the form in which they are kept is determined by the current state of the incremental compilation and linking. The fragments are annotated with "useful information" such as dependency graphs, cross references, and time stamps.

It should be observed that keeping a "precompiled data base of classes" doesn't by itself provide rapid compilations. Typically, a "compiled form" of a class will be *larger* than its source language form and experience with Simula shows that it can easily take something like 85% of the time needed to compile a class declaration simply to read a precompiled version of the class and integrate its information into the compilers tables. Pre-compiling header files may be essential for providing rapid dependency analysis, but it is likely that additional techniques will have to be employed to achieve compile speed. This might involve merging many headers into "standard environments" that can be stored in a form where they can be read back into the compiler without expensive unraveling of linked structures and adjustment of pointer values.

The C preprocessor will probably become a problem when designing a "smart" C++ environment. The key weakness of cpp in this context is that a .h file has no semantic meaning in C or in C++.

I conjecture that there will soon be several "smart" C++ environments. In my opinion, the single most important design criterion for such programming environments should be that each and every one of them should be capable to read in a C++ program on the "standard and dumb" form of .h and .c files as described in *The C++ Programming Language* and

† Using an interpreter during program development is an alternative. Such an interpreter would have to handle full context-dependent strong static type checking and would have to handle a mixture of compiled and interpreted code to be useful. I conjecture that the distinction between an interpreter and an incremental compiler isn't very significant in this context.

be able to emit a version of every program developed in the environment in this form. This is essential to maintain portability of C++ programs across environments.

Similarly, one must not *in general* assume an integrated system of the kind where the program development environment and the program execution environment are indistinguishable. It must be possible to "break out" a "final program" for use in a specific execution environment. It should also be observed that typically a C++ programmer cannot assume that there is a programmer of any sort (let alone a C++ programmer) available when the program is executed in its intended environment. This implies that to the end-user an invocation of a smart debugging environment and an ignominious core dump are completely equivalent.

### Incremental Linking and Loading

Incremental linking and loading are desirable for systems that are designed to allow a user to extend a running system. For example, in a CAD/CAM system it can be useful to allow a user to define a new kind of component and add it to a layout currently on the screen. Using traditional compile, load, and go techniques it is difficult to add the new component class to the layout program without having to re-create the "current layout" from scratch after re-compilation of the layout program. At best, there would be a noticeable disruption of service. A very similar example is a "user interface manager" that controls a user's interaction with application programs. Here, it can be useful to allow the user to add a new style of interaction but it would be most disruptive to stop, re-compile, and re-start the user interface manager to do so.

The key problem with incremental linking is to provide a way of adding a new class to a running program. To add a new class two problems have to be faced. It must be possible to define an interface to a yet-to-be-defined class and it must be possible to invoke operations on objects of the new class from the original program. The latter, incremental linking of functions, is a solved problem in most C environments. The former problem has a surprisingly clean and elegant solution in C++.

Traditionally, systems supporting incremental loading of classes have relied on some form of interpretation. That is, a user can try any operation on such an object and the object examines the request to decide if it can be accepted. If so, everything is fine and the operation is performed; if not, a run-time error ("method not found") occurs. In other words, static type-checking is impossible in such systems. This is unacceptable in many environments and clashes directly with the view that every object in a C++ program must be of a specific type so that the set of acceptable operations and their argument types are known at compile time.

The virtual function mechanism of C++ allows objects to be manipulated in a type-safe manner without knowing their exact type. This mechanism can be exploited to support type-safe incremental linking<sup>†</sup>. If the new class is derived from an already existing class with virtual member functions, the interface of the base class can be used to type-check calls to member functions of the derived class; the virtual function mechanism provides the binding between old code and the new.

Here is an example of a way the virtual function concept might be used to achieve this kind of incremental loading of classes. All one needs to do is to supply a base class with a function `new_object()` that takes a string argument designating a new derived class. For example:

<sup>†</sup> The technique for type-safe incremental linking described here is due to Jonathan Shapiro.

```

class shape {      // some base class
...
public:
    shape* new_object(const char *);
...

    virtual void rotate(int); // functions defining an interface
    virtual void draw();     // for all classes derived from shape
...
};

```

The `new_object()` function creates objects of classes that are undefined at the time when the program starts executing. For example:

```

shape* select(istream& input)
{
    shape* p;
    char class_name[MAX];
    // read name of shape class from 'input' into 'class_name'
    ...
    return p->new_object(class_name);
}

```

The class named by `class_name` must be a class derived from class `shape`. A object returned by `select()` can be used exactly like any other `shape`:

```

void some_function(int n)
{
    shape* p = select(cin)
    for (int i = 1; i < n; i++) {
        p->draw();
        p->rotate(180/n);
    }
    ...
}

```

All calls to the new object have been statically type checked and the overheads are exactly the same (i.e. very low) as the overhead on operations on objects of classes known when the program was originally compiled. Clearly, this differs from the usual "shape example" of virtual functions *only* in that the derived class might be defined *after* the program started executing.

When `shape::new_object()` is called, a class of the specified name must have been defined as derived from `shape`. Alternatively, `shape::new_object()` might prompt the user to write one. The functions for this derived class must then be compiled (if they were not already compiled) and the functions are linked to the running program using whatever technique for incremental linking and loading is available.

It is crucial that `new_object()` can find a constructor for the new class in the object file. A call of that constructor is used to create the new object (of the new class). This relieves `new_object()` of the responsibility of knowing the layout of objects of the new class, the size of such objects, the virtual functions re-defined by the new class, etc. To avoid loading a class twice `new_object()` maintains a table of known class names. Since the compiler didn't know about classes linked into the program in this way at the time when the program was originally written there cannot be static or automatic variables of such classes except in parts of the program written and linked in later yet. Several (overloaded) versions of `new_object()` can be provided for a base class to give the effect of several constructors. Naturally, `new_object()` functions would employ standard library functions to do most of their work.

Complete re-compilation is only necessary when a new class with a hitherto undefined calling interface needs to be added. This method combines flexibility with efficiency and type safety in a very appealing manner.

## Object I/O

Some problems are traditionally handled by providing "utility routines" with detailed information about the structure of all objects in a system. Examples of such utility programs are debuggers that allow a programmer to inspect the representation of objects in the program being debugged, routines that print objects, routines that write objects to and from disc, and routines that transmit and receive objects over a communication line.

Such routines traditionally rely on "maps" describing the layout of objects. Such maps are produced by "the system" specifically to allow objects to be inspected at run-time. For C++, such a map would describe a class in such a way that the names, types, and relative position of every member of a class object would be known at compile time. In addition, information about the class hierarchy and maybe about friends would be needed. Naturally, a C++ compiler could produce such maps. However, it might be better to do without such maps as far as possible. In essence, any use of such a map represents a violation of the encapsulation of objects and is at odds with idea of a system as a set of self contained objects each with a single well-defined checked interface. In other words, to a routine using such a map every object is just global data

and a major design aim of C++ is to get away from programming styles relying on anarchic use of run-time global data. The use of such maps should be avoided where reasonable alternatives exist.

The alternative to using such maps is to provide classes with member functions that provide operations necessary to compose inspection, I/O, communication, etc., without introducing functions treating objects as mere data.

Consider the problem of producing a copy of an object of a class X in a form that can be transmitted over a wire and reconstituted at the other end:

```
class X {  
    int a;  
    char* b;  
    Y c;  
public:  
    ...  
}
```

Fundamentally, the problem breaks down into two parts: to transmit the members and to transmit the information that these members were part of an X. There are various techniques for expressing the latter and also various techniques for coping with the nasty details such as linked structures. These can be applied independently of the approach taken to transmit the members, so this set of problems can be ignored here.

Clearly, any approach that respects the encapsulation properties of class X must involve the introduction of a member function to transmit the members. For example:

```
class X {  
    int a;  
    char* b;  
    Y c;  
public:  
    xmit(ostream& s); // encode members  
                      // and write them to output stream "s"  
    ...  
}
```

Basically, xmit() can be written by applying xmit() to each of the members:

```

X::xmit(ostream& s)
{
    // write transmission header
    xmit(s,a);
    xmit(s,b,ZERO_TERMINATED); // xmit b as a zero terminated string
    c.xmit();
    // write transmission trailer
}

```

This way object output is defined as memberwise and recursive. Naturally, this depends on each of the members having an `xmit()` operation. Keith Gorlen's OOPS library is an example of this technique in the context of C++.

Writing such a set of `xmit()` functions is easy, so easy in fact that it can become tedious. Given a set of `xmit()` routines for the built-in types (taking care of C problems such as the inability to distinguish a pointer from a vector in general) the compiler or a similar tool can write the `xmit()` functions in cases where the user hasn't specified one and where the default rules for constructing an `xmit()` function apply.

A tool that generates member functions in this way and a tool that generates an object map with a function that takes advantage of it are largely equivalent. The difference only appears if object maps become an integral part of a system used by large numbers of application level programs. Object maps and mechanisms for automatically generating large numbers of functions are both mechanisms for applying global operations to a system and should be used sparingly.

### Persistent Objects

A mechanism that allows an object to be stored and restored can be used to implement a notion of persistent objects. In addition to the basic I/O operations a naming scheme must exist for both classes and individual objects. Such schemes and facilities for making them available to application programs are relatively easy to provide.

Consider writing an object of class `X` out from one program and reading it into another. The object I/O is handled using the techniques described above. In the case of a persistent object, the object is read in by a constructor, the "permanent copy" is if necessary updated appropriately from the copy in temporary store, and the final update of the permanent copy is guaranteed by `X`'s destructor.

This leaves only one problem: What if the program that is trying to read in the object doesn't know class `X`? This problem can be handled using the technique for incremental linking and loading described above provided `X` is derived from a known class `B`. In that case, the program simply looks up `X` in some directory of classes, requests loading of class `X` using something like `new_object()`, and continues reading in the object using a constructor for `X`.

### Automatic Storage Management

C++ inherited from C a view of objects where an object can be of one of several storage classes, each with its own rules for allocation and de-allocation. This provides important efficiencies in both time and space but can be a burden on the programmer.

The management of static and automatic objects in C++ is automatic in the sense that storage is managed without intervention from the programmer and that constructors and destructors are called appropriately. The same applies to objects that are members of aggregates. Objects allocated on the free store must be explicitly de-allocated before the space can be re-used. The manual aspects of this are a source of complexity in programs and a common source of nasty bugs in C programs. On the other hand, the overheads of a perfectly general free storage allocator and de-allocator are typically quite noticeable for programs relying on it.

In C++, a large fraction of objects on the free store is typically managed as "secondary data structures" of other objects. That is, they are allocated by a constructor for some other object and de-allocated by the destructor of that object. The management of such "secondary objects" is so simple that it does not affect the complexity of a program noticeably. This reduces the complexity of handling objects on the free store considerably naturally, the number of errors goes down too. Furthermore, where class specific allocators and de-allocators are employed the overheads of free store use drops to insignificance.

The problem of managing storage "automatically" is now reduced to minimizing the effort of the programmer in handling the remaining set of "primary objects". This, however, can still be a problem when writing general purpose libraries where the storage class of objects handed to utility functions are not generally known. For example, a user inserts a pointer to an object into a container. Who is responsible for de-allocating the object (and ensuring that its destructor is called)?

The user?

The container?

The system?

In general, the user cannot do the de-allocation since the user does not know if the container has stored a pointer to the object somewhere. If the container has done that and the user deletes the object chaos will erupt. Similarly, the container cannot delete the object because it does not know what the user might be doing with that object.

The system, provided there is a system, can look at all of memory and delete the object only when it becomes unreferenced. This garbage collection approach is expensive†, does not take advantage of the structure of C++'s memory as organized by the storage classes, and is hampered by the fact that C++ wasn't designed with garbage collection in mind.

The actual cost of garbage collection is notoriously hard to estimate. The notion of "managed store" pervasively affects both the design of language features and the implementation strategies of languages designed with garbage collection in mind. Similarly, the notion of storage classes pervasively affects C++ and the implementation strategies for C++. For example, it is not possible to estimate the cost of garbage collection simply by comparing a run of a program in a small memory (where the garbage collector is activated) with a run of the same program on the same input in a memory large enough to ensure that the garbage collector isn't activated.

In languages designed for garbage collection such as Lisp and Clu, an object of a non-primitive type is accessed through a fixed sized handle. In C++, all objects may be allocated and accessed directly. Compared with a language designed to minimize allocation costs and memory accesses, such as C or C++, a "garbage collection language" spends cycles simply getting to the objects, allocating objects, and maintaining the information necessary to do garbage collection should it become necessary.† This implies a less compact representation

† This discussion of garbage collection assumes that C++ is used on a fairly traditional machine. C++ was designed for such traditional architectures. The fundamental assumption is that relatively simple, relatively traditional architectures will remain more cost effective than more complicated architectures which support higher-level concepts such as garbage collection directly.

The volume of production of traditional "general purpose" machines is likely to far outstrip the production of more specialized architectures for the foreseeable future. This assures a heavy investment in keeping such traditional hardware efficient. Simpler architectures typically also have a lower design time thus ensuring that a traditional architecture will on average be ahead in manufacturing technology.

Should a situation arise where this fundamental assumption does not hold the question of garbage collection for C++ will be re-evaluated. This situation can of course arise locally, making garbage collected C++ feasible on a particular range of machines. In this case, portability issues become interesting.

† There is unfortunately very little hard information available. The comparisons of "efficient" garbage collected languages such as compiled Common Lisp and "efficient" non garbage collected languages such as C seem to point to a factor of 3 to 5 difference. Sloppy use of garbage collection can easily lead to factors of 10 to 50, but sloppy choice of algorithms or sloppy use of C-style free store can also lead to drastic increases of costs. In both cases, the nastiest aspect of such overheads is that the programmer typically doesn't know if they are fundamental or simply the result of lack of understanding or bad engineering. It is often easier for the programmer simply to assume the former than to try to discover if there really is a problem.

of objects in memory; that is, in most cases a garbage collection language will require more memory to represent the same set of objects than does C++.

To complicate matters further, it is (obviously?) difficult to measure the cost of *not* using garbage collection, since the code related to memory management will be scattered throughout the system and the design of the system is typically done so as to minimize the effort needed to manage store.

Naturally, run-time and memory isn't all one needs look at when estimating cost. Ideally, the cost of using or not using garbage collection should be considered in terms of design time, debugging effort, maintenance effort, portability, run-time, store usage, complexity of supporting hardware, complexity of program development, complexity of run-time support software, complexity of compilers, etc.

It seems unlikely that simply applying the garbage collection techniques perfected for languages designed with garbage collection in mind to C++ would make C++ into a good garbage collected language. It might be done, but I suspect that since the fundamental design choices for C++ were made assuming the absence of garbage collection, a C++ garbage collector would be less efficient than, say, a good Lisp or Clu garbage collector. Worse, I suspect that a garbage collected C++ would fail to deliver the low-level performance people have come to expect from C++.

A more appropriate course of action seems to be to continue the C and C++ approach to increase the programmers set of choices for allocating objects with something that could be seen as a "garbage collected" storage class. This would simply allow some objects to be used with the convenience of *automatic* (stack) storage, but with unrestricted life times like objects on the C++ *free store* (dynamic storage). Unfortunately, this is not simple and especially not simple given the obvious requirement that the use of such "managed" objects should not be expensive compared with other C++ objects or compared with the use of objects in a garbage collected language. To make matters worse, the extra complexity to the user *must* be minimal to make the scheme attractive and it would also be nice if the scheme did not require modification to the compilers.

Surprisingly enough, such schemes appear to be feasible. There are at least two possible approaches. A storage manager can use constructors, destructors, etc. to keep track of

- all "managed objects", or
- all pointers to managed objects.

Jonathan Shapiro's "counted pointer" classes<sup>9</sup> take the latter approach. A counted pointer is an object that acts like a pointer. Counted pointers for an object are "counted" and when the last counted pointer to an object is destroyed the object is freed. Overloading of operators usually denoting indirection such as unary \* and -> is essential for this approach.

The following program makes and manipulates sequences of integers. A ListCell\_CP is a counted pointer to a list cell:

```
main()
{
    ListCell_CP p = makeSeq(10);           // p = (0 1 2 3 4 5 6 7 8 9)

    ListCell_CP temp = select(&p,isPrime); // temp = (1 3 5 7)
    cout << "\nPrimes: " << *temp << "\n";

    p = select(&p,isOdd);                   // p = (1 3 5 7 9)
    cout << "\nOdds: " << *p << "\n";

    temp = upTo(10,isPrime);               // temp = (1 3 5 7)
    cout << "\nPrimes: " << *temp << "\n";
    cout << "\nOdds: " << *upTo(10,isOdd) << "\n"; // print (1 3 5 7 9)
}
```

Naturally, several libraries were included to compile and run this program. The main point about it is the absence of end-user-specified memory management code. Note for example,

the assignment to `p` in the statement that selects odd numbers from `p`. All memory is reclaimed and if possibly re-used automatically. In particular, all memory is reclaimed before the completion of the program. No modifications to the compiler were required to achieve this. The output was:

```
Primes: 1 3 5 7
Odds: 1 3 5 7 9
Primes: 1 3 5 7
Odds: 1 3 5 7 9
```

For some applications an extremely crude and effective approach to memory management is possible: never delete anything. This technique can lead to spectacular improvements for programs that simply doesn't need as much memory as is available on the machine. C++'s compact representation of data and direct access to data makes it well suited to this. Naturally, systems intended for runs of uncertain duration or uncertain memory requirements cannot use this approach; neither can general purpose library functions and classes.

#### 4 Design Methods

It is not always clear how best to take advantage of a language such as C++. Significant improvements in productivity and code quality have consistently been achieved using C++ as "a better C" with a bit of data abstraction thrown in where it is clearly useful. However, further and noticeably larger improvements have been achieved by taking advantage of class hierarchies in the design process. This is often called object-oriented design and this is where the greatest benefits of using C++ have been found. For my views of what "object-oriented programming" and "data abstraction" are see reference 11.

Two levels of problems face us here. Firstly, most of the experience relating to object-oriented design is either not written down or only written down in a form that makes it inaccessible to novices. Hence, education is a major concern; the bottleneck is the low number of genuinely experienced people and the fact that they are typically frantically busy.

After solving the problem of getting experience in object-oriented design put on paper and in other ways made accessible we must face a harder problem. There exists (to my knowledge) no specific "object-oriented design method" with associated conventions, notations, standardized terminology, standardized procedures, and (ideally) tools. Such methods are needed to allow *large* numbers of designers and programmers to cooperate. Using a standard non-object-oriented design method simply doesn't deliver the results that have been shown to be attainable from C++.

Design methods that have the concepts of class, hierarchy, inheritance, and statically typed interface as integral parts are badly needed. The requirement that a design method should cope with all of these concepts implies that a designer must keep in mind that C++ isn't

- Smalltalk
- Lisp
- Ada
- (just) C

etc. Conversely, those languages are not C++ and a C++ programmer switching to one of those for a project should not uncritically try to impose C++ notions on them. It is a well known problem that someone's first program in a new language often simply is a program in the programmer's previous language written using the new language's syntax. A very similar trap exists for designers.

Naturally, a design method must be chosen and applied with care. Projects with different aims, different organizations, different magnitudes of problems, and different skills of the people involved require differences in design method and in the style of its use. There is no and there can be no simple set of rigid rules and conventions that turns novices

into first rate programmers and designers. Only education and experience can do that. On the other hand, a set of rigid rules and conventions can completely cripple perfectly capable programmers and designers.

There is no substitute for experience, taste, and insight when designing a complex system; a design method should enhance these attributes of a designer by reducing trivia, mistakes, and oversights. Typically, a design method is no substitute for experiment either.

Sometimes it will be necessary to adjust the organizational structure to make best use of a design method. For example, it may often be necessary to make adjustments to ensure that the sharing of libraries and the creation of libraries that can be shared are genuinely encouraged. Similarly, a change of the unit of job assignment and in the unit of ownership of code (that is, responsibility for maintenance) may be needed when moving from a traditional design method to a C++ object-oriented style of design.

## 5 Danger Areas

There are two major dangers that must be faced when considering the evolution of C++†:

- [1] Extend the language fast to please users. This could cause rampant featurism and lead to increases in:

- compiler sizes and compilation times;
- the time needed to learn C++;
- the time needed to port a C++ implementation;
- run-time and size of C++ programs.

- [2] Freezing the language specification at the current stage and accepting new features, libraries, and tools extremely slowly to ensure stability.

- Language fragmentation; every supplier would add a few "essential features;"

- Rampant featurism as suppliers compete to produce better C++ supersets;

- Loss of portability of programs between different C++ implementations.

Avoiding all of these problems simultaneously isn't easy. A most important rule is not to damage the ability to write very efficient low-level C++ programs. A bit of this kind of code is essential to almost every large application and almost all interesting libraries have a few key operations where efficiency is essential to make the use of the library an acceptable alternative to "hand optimized" code. The most obvious ways of damaging the basic efficiency of C++ would be to introduce new features that depended on either garbage collection or interpretation of dynamic type information in such a way that every program and every part of a program had to help carry the cost of the "advanced" features. This would cause C++ libraries to become less affordable and produce even greater demands for specialized language extensions. This again would cause C++ to become harder to port and probably lead to a greater number of dialects.

It is essential to maintain link compatibility with C program fragments and important to make calls to and from routines written in other languages as simple and efficient as possible.

## 6 Conclusions

C++ can be developed into a better language for library design and use. This probably means adding a parameterized type mechanism and an exception handling mechanism. C++ should be extended primarily as a tool for building libraries and not by adding features for coping with specialized applications. Such applications should be supported through libraries. Efficiency of C++ primitives (compared to C primitives) must be maintained.

† Assuming, of course, that the most obvious source of incompatibilities, an imprecise or incomplete language specification, is taken care of. The C++ reference manual<sup>10</sup> isn't sufficient as an implementor's guide. To complement that a commentary for implementors is being developed together with a C++ verification suite. The work of the ANSI C committee is also of great help.

C++ programming environments with incremental compilation and incremental linking and loading would be of great use to programmers, but care must be taken to ensure that program source can be cost-effectively transferred between different such environments. The C++ mechanisms for specifying statically typed interfaces and its encapsulation mechanisms shouldn't be undermined by the introduction of dynamic type information except as a last resort.

We need design methods that allow a designer to take advantage of the data abstraction and object-oriented features of C++.

## 7 Acknowledgements

This paper owes most to people who have been pushing the limits of C++ in new application areas and tried out techniques that will only become feasible on a large scale as the C++ language and its associated tools matures further; notably, Keith Gorlen, Andy Koenig, and Jonathan Shapiro. Much has also been learned from discussions with Tom Cargill, Jim Coplien, Brian Kernighan, Doug McIlroy, and with C++ users acquainted with the Smalltalk and Lisp integrated environments.

This paper extrapolates from techniques already used and tools already being considered or built for C++. This implies that just about every problem mentioned here is actively being addressed somewhere.

Thanks to Andy Koenig for inviting me to give a talk with the given title "What will C++ look like two years from now?" that eventually became this paper.

## 8 Postscript

This is *not* a paper I enjoyed writing. It describes things that *might be*, not things that *are*; *possibilities*, not *experiences*. It seemed a useful paper to write, though, because each of the possibilities described will be explored many times in many contexts and if each is considered only in isolation a terrible mess will result.

Each individual part described is reasonably straightforward to design and given a bit of experimentation and common sense it can be gotten "reasonably right." Combinations do not appear to be so. Combine any two parts together, say, parameterized types and exception handling or persistence and incremental linking, and the possibilities for making serious mistakes increase dramatically and the feasibility of conducting reasonably sized experiments decreases.

I consider no part of this paper self evident or even just un-controversial. For each part of a C++ system, there are several alternatives to the design outlined here and many ways of turning that vague design into a real tool. I hope this paper can be the starting point for many discussions and designs. It should remind designers of individual parts of a C++ system about the potential of a complete system, warn about the potential complexity of such a system, and point out that theirs isn't going to be the only such C++ system.

## 9 References

- [1] DeMichiel, Linda G. and Gabriel, Richard P.: *The Common Lisp Object System: An Overview*. Proc. ECOOP'87. Lecture Notes in Computer Science, Springer Verlag. Vol 276. June 1987.
- [2] Goldberg, A. and Robson, D.: *Smalltalk-80: The Language and its Implementation*. Addison-Wesley 1983.
- [3] Ichbiah, J.D. et.al.: *Rationale for the Design of the Ada Programming Language*. SIG-PLAN Notices, June 1979.

- [4] Gorlen, Keith: *An Object-Oriented Class Library for C++ Programs*. Software Practice and Experience, December, 1987.
- [5] Koenig, Andrew: *Associative Arrays in C++*. AT&T Bell Labs Internal Report.
- [6] Liskov, Barbara et. al.: *Clu Reference Manual*. MIT/LCS/TR-225, October 1979.
- [7] Shopiro, Jonathan: *Lists and Strings for C++*. AT&T Bell Labs Internal Report.
- [8] Shopiro, Jonathan: *Extending the C++ Task System for Real-Time Applications*. Proc. USENIX C++ Workshop, Santa Fe, November 1987.
- [9] Shopiro, Jonathan: *Counted Pointers An Automatic Storage Management System for C++*. To be written.
- [10] Stroustrup, Bjarne: *The C++ Programming Language*. Addison-Wesley, 1986.
- [11] Stroustrup, Bjarne: *What is Object-Oriented Programming?*. Companion Paper.
- [12] Stroustrup, Bjarne: *The Evolution of C++: 1985-1987*. Companion Paper.
- [13] Weinreb, D. and Moon, D.: *Lisp Machine Manual*. Symbolics, Inc. 1981.

# A Set of C++ Classes for Co-routine Style Programming

*Bjarne Stroustrup  
Jonathan E. Shopiro*

AT&T Bell Laboratories  
Murray Hill, New Jersey 07974

## ABSTRACT

Some programs are most naturally expressed as a set of relatively independent activities communicating to achieve a common goal. Each activity, here called a *task*, has its own locus of control, a program to execute, and its own private data. Tasks can communicate by explicit sharing of data, by messages, or by data pipes.

This paper describes C++ classes for a range of styles of multi-programming techniques in a single language, single address-space environment. Each task is an instance of a user-defined class derived from class `task`, and the program of the task is the constructor of its class. A task can be suspended and resumed without interfering with its internal state. Class `qhead` and class `qtail` enable a wide range of message passing and data buffering schemes to be implemented simply.

The task system can be used for writing event driven simulations. Tasks execute in a simulated time frame presented by the variable `clock`, and objects of class `timer` provide a convenient and efficient facility for using the clock.

The implementation and use of these concepts rely heavily on the idea of derived classes. Familiarity with the C++<sup>[1]</sup> language would be an advantage for the reader.

## 1. INTRODUCTION

Some programs are most naturally expressed as a set of relatively independent activities communicating to achieve a common goal. Such activities, here called *tasks*, must be able to execute in parallel with each other and communicate through means convenient to the chosen style of task usage.

Facilities for multi-thread computation can be provided in the semantics of a language, as is done in Concurrent Pascal<sup>[2]</sup> and Mesa<sup>[3]</sup> or a language without such facilities can be augmented using special run-time support systems and library functions, as has been done for BCPL<sup>[4]</sup> and C.<sup>[5]</sup> The use of C classes to implement tasks represents an intermediate approach pioneered by Simula67.<sup>[6]</sup>

The tools presented here<sup>†</sup> provide the basic facilities for several styles of multi-thread programming in a single language, single address-space environment. The underlying facility is a simple and efficient tasking system with non-preemptive scheduling. That is, a task will only be suspended on its own request, so no "system policy" can be enforced without

<sup>†</sup> The original version of this paper was written in 1980 by B. Stroustrup and revised in 1982 by him. Since then both the task library and C++ (then known as "C with Classes") have changed substantially, but the interface to the task library has been left intact. This has allowed old programs to run with new versions of the library, but has prevented any updating of the style of the interface, which does not conform to current tastes.

This version of the paper has been revised by J. E. Shopiro to reflect the present state of affairs. I have added a few notes (in sans-serif type) where changes have been significant, and have made numerous syntactic changes, etc., without further comment.

the cooperation of all tasks. In contrast to pure co-routine systems, however, the task system provides a framework for processor sharing and communication between tasks.

The task system is intended for applications, like event driven simulations, where tasks are used to express a quasi-parallel structure for a single program. For this class of applications a concept of simulated time is implemented. A unit of simulated time can represent any amount of real time, and it is possible to compute without consuming simulated time. A few simple random number generating classes and a histogram class for data gathering are also provided. The task system is not intended for handling real parallelism of some underlying real-time system. Consequently, no facilities are provided to map interrupts and other real-time events into the concepts provided by the task system.

The current version of the task library<sup>[7]</sup> has a new degree of extensibility, so that it is now possible to write a class that represents an interrupt or signal that can be waited for.

Implementations of the task system have been used for about eight years on UNIX<sup>††</sup> and other operating systems on 3B2, 3B20, VAX, and Motorola 680x0 hardware.

In the following sections the task library will be described in some detail, and examples of its use will be given. The classes used in the task system are presented. This allows a detailed and specific discussion of the concepts involved, but it unfortunately also implies that some concepts cannot be explained in detail where they are first mentioned.

## 2. TASKS

The publicly accessible functions and data of class `task` look like this:<sup>1</sup>

```
class task : public sched
{
public:
    task(char* name=0, int mode=0, int stacksize=0);
    ~task();
    task* t_next;
    char* t_name;
    int waitvec(object**);
    int waitlist(object* ...);
    void wait(object*);
    void delay(int);
    int preempt();
    void sleep(object* t =0);
    void resultis(int);
    void cancel(int);
};
```

The base class, `sched`, is responsible for scheduling and for the functionality that is common to tasks and timers (described below). The public part of its declaration is:

<sup>††</sup> UNIX is a registered trademark of AT&T.

1. Many of the member functions are inline, but their definitions are not shown here to prevent clutter. Class `task` is derived from class `sched` which is derived from class `object`. Class `object` is a simple base class used by most classes in the task system. It contains some of the pointers used by the task system's internal "house-keeping". Class `object` is presented in appendix A.

```

class sched : public object {
public:
    sched();
    void setclock(long);
    long rdtime();
    int rdstate();
    int pending();
    void cancel(int);
    int result();
};

```

Class sched is used strictly as a base class: that is, only instances of derived classes are created.

A task is a locus of control, a virtual processor. It too can only be used as a base class. A task executes the program supplied as the constructor of the derived class.<sup>2</sup> The most basic feature of a task is that it can be suspended and later resumed so that several tasks can run in quasi-parallel. Most member functions of class task are conditional or unconditional requests for suspension.

A task can be in one of three states:

- RUNNING** The task is executing instructions or it will be scheduled to do so without further intervention from other tasks.
- IDLE** The task is not in the RUNNING state, but it can be transferred to the RUNNING state by some suitable action. That is, it is waiting.
- TERMINATED** The task has completed its work. It cannot be resumed, but its result can be retrieved.

The function `sched::rdstate()` returns the state.

A simple example of the use of tasks is where one task creates another to run in parallel with itself. Later the creator can obtain the result produced by the "secondary" task. For example, a task which counts the number of spaces in a string could be declared. First a class Spaces must be declared.

```

class Spaces : public task
{
public:
    Spaces(char*);
};

```

In the case of class Spaces the declaration is trivial. It states that Spaces is derived from class task so that each object of class Spaces becomes an independently scheduled entity. The program for the task is provided by its constructor.

2. The class may have other member functions, of course, which may be called by the constructor or by any other function according to the usual rules of C++.

```

Spaces::Spaces(register char* s)
{
    register int    i = 0;
    register char   c;
    while (c = *s++)
        if (c == ' ') i++;
    resultis(i);
}

```

This function counts the spaces in its argument string and returns the result using the class task function `resultis()`. A task of class `Spaces` can now be created and used like this:

```

Spaces    ss("a line with four spaces");
// ...
count = ss.result();

```

When an object of class `Spaces` is created, like `ss` here, its constructor becomes a new task that runs in parallel with the task<sup>3</sup> that created it. A task can "return" an integer<sup>4</sup> value using the function `task::resultis(int)`. The task then becomes `TERMINATED` and the value is available for examination by the function `sched::result()`. That is, in this example `ss` will call `resultis()` with the argument 4 which will be returned from `sched::result()` to the parent task. If a task calls `result()` for another task which has not yet completed the calling task will be suspended. After the other task finishes the call to `result()` in the waiting task will return. A task waiting for another to complete is `IDLE`. If a task calls `result()` for itself it will cause a run time error.<sup>5</sup>

A task cannot return a value using the usual function return mechanism; it must use `resultis()`. This function puts the task into the `TERMINATED` state from which it cannot be resumed.

### 3. QUEUES

A *queue* is a type of storage that is organized so that objects are retrieved from it in the order in which they were inserted into it. A queue has a *head* from which data is retrieved and a *tail* where data is inserted. With a little elaboration this basic type of data structure makes an excellent inter-task communication facility.

There is no "class queue" available to a user. Instead, the two classes `qhead` and `qtail` provide the services needed. There is a function `qtail::put()` which adds an object to the *tail* of a queue and a function `qhead::get()` which retrieves an object from the *head* of a queue. This allows explicit separation between the source and the recipient of data. The public part of the declaration of class `qhead` looks like this:

- 
3. When the first task is created, `main()` automatically becomes a task itself.
  4. It is a fairly simple job to add a new kind of task that returns some other datatype.
  5. The handling of run time errors will be described below.

```

class qhead : public object
{
public:
    qhead(int =WMODE, int =10000);
    ~qhead();
    object* get();
    int putback(object*);
    int rdcnt();
    int rdmode();
    int rdmax();
    void setmode(int);
    void setmax(int);
    qtail* tail();
    qhead* cut();
    void splice(qtail *);
    int pending();
    void print(int, int =0);
};

```

A queue can be created like this:

```
qhead    qh;
```

To obtain a qtail for an existing queue execute tail() for its head:

```
qtail*    qtp = qh.tail();
```

The queue could now be used as a one way inter-task communication channel by giving its head and tail as arguments to two new tasks, Producer and Consumer:

```

Producer  pp(qtp);
Consumer  cc(&qh);

```

The producer task pp can now put() objects to the tail of the queue (denoted by the pointer qtp) and the consumer task cc can get() those objects from its head (denoted by the pointer &qh). The function qtail::put() takes a pointer to a class object as argument, and qhead::get() returns such a pointer. Unless the user has specified otherwise a task executing qhead::get() will be suspended temporarily if the queue is empty.<sup>6</sup> After another task executes put() on the associated queue tail the suspended task will be resumed. Similarly a task executing qtail::put() on a full<sup>7</sup> queue will be suspended until some other task removes data from the queue.

The objects transmitted through a queue must be of class object or of some class derived from it. Class object (described in appendix A) is provided by the task system, and it is up to the programmer to define types of objects suitable for each application.

In the current version of the task library qhead and qtail have the form of user extensions, but in the original version they were built in. Since extensibility was limited, the supplied classes had to support a wide range of programming styles. Thus they may seem "feature-rich". The new organization makes it easy to provide new kinds of queues and other forms of task interaction.

6. Thus qhead::pending() returns 1 if the queue is empty and 0 otherwise. Correspondingly, qtail::pending() returns 1 if the queue is full and 0 otherwise.

7. The default maximum size for a queue is 10000. That is, the queue can hold up to 10000 pointers to objects. It does not, however, pre-allocate space.

### 3.1 A Server Example

As an example of the use of tasks and queues we will define a "server" task that receives requests for service in the form of messages on a queue, handles the requests and returns replies on other queues. One could define a class `Message` as follows:

```
class Message : private object
{
public:
    int      r_operation;
    int      r_arg1;
    int      r_arg2;
    qtail*   r_reply;
};
```

A message, that is an object of class `Message`, describes an operation `r_operation` that is to be performed by the recipient of the message. Arguments for this operation can be passed as `r_arg1` and `r_arg2`, and the result of the operation is to be returned as a message on the queue denoted by `r_reply`.

A server for these messages can be defined as follows:

```
class Server : public task
{
    Server(qhead *);
};

void Server::Server(qhead* in)
{
    for (;;) {
        Message* req = (Message *) in->get();
        queue*   reply = req->r_reply;
        int      res = VALUE;
        int      val;
        switch (req->r_operation) {
        case PLUS:
            val = req->r_arg1 + req->r_arg2;
            break;
        case MINUS:
            ...
        default:
            res = ERROR;
        }
        req->r_operation = res;
        req->r_arg1 = val;
        reply->put(req);
    }
}
```

This style of server has proved useful in many contexts. In particular, it is the backbone of many "message-based systems". In this particular example a server, that is an object of class `Server`, and the queue on which it depends can be declared:

```
qtail*   rq = new qtail;
Server*   ser = new Server(rq->head());
```

Other tasks can now send a request to this particular server through `rq`. For example:

```
qhead     rply;
qhead*    rply_to = rply.tail();
Message*   mess = new Message;
```

```

mess->r_operation = PLUS;
mess->r_arg1 = 1;
mess->r_arg2 = 2;
mess->r_reply = reply_to;

rq->put(mess);
mess = (Message *) rply->get();
if (mess->r_operation == ERROR) error();

```

### 3.2 More about queues: mode and size

A queue head has a *mode* that controls what happens when `get()` is executed on an empty queue. In *EMODE* this causes a run time error. In *ZMODE* it will cause `get()` to return the *NULL* pointer instead of a pointer to an object. In *WMODE* a task executing a `get()` on an empty queue will wait on that queue until the queue becomes non-empty. Unless the user specifies the mode explicitly a queue head will be in *WMODE*. The function `qhead::rdmode()` returns the current mode and `qhead::setmode()` can be used to change it.

As mentioned above a queue also has a maximum size. This can be changed using `qhead::setmax()`, and read using `qhead::rdmax()`.

The mode and maximum size for a queue can also be specified when the queue is created. For example:

```

qhead    Q1(ZMODE, 10);
qhead*   QP2 = new qhead(EMODE, 64*BUFSIZE);

```

The public part of the declaration of class `qtail` is similar to that of class `qhead`. The two classes complement each other, and together they provide a representation of the general idea of a queue:

```

class qtail : public object
{
    // ...
public:
    qtail(int = WMODE, int = 10000);
    ~qtail();
    put(object*);
    int    rdspace();
    int    rdmax();
    int    rdmode();
    qtail* cut();
    void    splice(qhead*);
    qhead*  head();
    void    setmode(int m);
    void    setmax(int m);
    int    pending();
    void    print(int, int = 0);
};

```

A queue tail's mode controls what happens on queue overflow in the same way as queue head's mode controls what happens on queue underflow. For example, when a task executes `put()` on a full queue where the queue tail is in *WMODE*, then that task will be suspended until the queue is no longer full. The modes of a queue's head and tail need not be the same.

Similarly the maximum number of objects which can be on a queue can be examined by `rdmax()` and changed by `setmax()`. Decreasing the maximum below the current number of objects on the queue is legal. Doing this simply implies that no new objects can be put on the queue until the queue has been drained below the new limit.

Qhead::rdcount() returns the current number of objects in a queue, and qtail::rdspace() returns the number of objects which can be inserted into a queue before it becomes full.

Qhead::putback() puts its argument back at the head of the queue, that is

```
qhead    qh(WMODE,10);
object*   oo = qh.get();
qh.putback(oo);
oo = qh.get();
```

will assign the same object to oo twice. Putback() has proved to be a useful function in many systems in the past, and it also allows a queue head to operate as a stack. When putback() is used, the task executing it competes for queue space with tasks using put() on the queue's tail. A putback() to a full queue causes a run time error in both EMODE and WMODE. In ZMODE it returns NULL.

#### 4. MORE ABOUT TASKS

When a task is created it can be given three arguments. The first is a character string pointer which is used to initialize the class task variable t\_name. This name can be used to provide more readable output and does not affect the behavior of the task. The string denoted by the pointer will not be copied. The t\_name is used by the debugging aids and error reporting functions described below. The other two class task arguments are tuning parameters and will be described below. If an argument is NULL a system default will be used. For example, we could have given each Server task a name like this:

```
class Server : public task
{
    Server(char*, qhead *);
};

void Server::Server(char* name, qhead* in)
: (name) // argument for Server's base class task
{
    // ...
}

Server my_name_is_fred("fred", qhp);
```

Task::sleep(object\* =0) suspends the task unconditionally without specifying what is supposed to cause it to be resumed.

If an argument is given to task::sleep(object\* =0) which is a pointer to a pending object, the task will be *remembered* by the object, so that after it is no longer pending, the task will be resumed.

Task::cancel() puts a task into the TERMINATED state and sets the return value just like resultis(). However, cancel() does not invoke the scheduler so that one task can terminate another without losing control itself.

The pointer<sup>8</sup>

```
task*    thistask;
```

8. The original task package had a number of global variables, including thistask, task\_chain, and clock. They are now all macros which expand to inline functions that return the values of private static variables. Thus programs that just read the values will be unaffected, but programs that try to set them (which was always illegal) will fail to compile.

denotes the currently active task. If no tasks have been created its value is 0. It is illegal to assign to `thistask`. The use of `thistask` enables the class `task` functions to be used from external functions without explicit passing of the current task's `this` pointer.

The pointer

```
task* task_chain;
```

is the start of a chain of all tasks. In the following loop `t` points to every task in turn:

```
task* t;
for (t=task_chain; t; t=t->t_next) ;
```

It is not possible to have only one task. Therefore, when the first task is created in a program another task is implicitly created. Its name is `main` and its code is the original `main()` function. It can be suspended and resumed like any other task. Please remember that a return from `main()` terminates a C program. If the "main" task should be terminated when there are other tasks which should be left running, then `resultis()` can be used. For example,

```
thistask->resultis(0);
```

can be executed in `main()`. The program will then run on until no more tasks are or can become `RUNNING`.

It is illegal for a task to return. Always call `resultis()` instead of `return`, and never just "drop out of the bottom" of a task. Unless a task contains an infinite loop so that it will never terminate place a call of `resultis()` at the end of its body.

The task system does not provide a garbage collector. It is left to the programmer to ensure that pointers to deallocated store are not used.

## 5. WAITING

Functions like `task::result()`, `qhead::get()`, and `qtail::put()` each provide a way of waiting for one single specific event to happen. More general facilities are sometimes needed.

When an object must be waited for, we say it is *pending*. For example,

- A queue head whose associated queue is empty is pending because if a task calls `get()` for it, the task must wait until some other task puts some data in the queue,
- Similarly, a queue tail whose queue is full is pending because a `put()` must wait, and
- A task that has not terminated is pending because its result is not available.

Each class derived from `object` may have its own definition of the virtual `pending()` function. An object may have several operations that could suspend the calling task, but it can have only one definition of `pending()`. Therefore (for example) it is not possible to combine a queue head and a queue tail into a single object, because the former is pending when its queue is empty, and the latter when its queue is full. New kinds of objects, with new kinds of interaction can be added to the task library, with the fundamental requirement being a definition of `pending()` for the new datatype.

`Task::wait(object*)` provides a way of waiting on an arbitrary object. If the argument points to a pending object, the calling task will be suspended until the object is no longer pending. If the argument is not pending the caller will not be suspended at all. For example, if `taskp` is a pointer to a task then

```
wait(taskp);
```

will suspend the task executing it until the task denoted by `taskp` finishes.

Each class derived from class `object` which is ever going to be "waited on" must have rules specifying under which conditions a task executing a `wait()` for it will be resumed. The rules for class `task`, `qhead`, and `qtail` have been stated.

The conditions for wakeup are reflected in state changes in the objects, and are not just transitory unrecorded signals. For example, if a task executes a `wait()` for a non-empty `qhead` it will immediately continue, that is the condition for returning from a `wait()` for a `qhead` is that the queue is non-empty, not a brief state change from empty to non-empty. Rules of this type simplify programming considerably by eliminating race conditions.

When the state of an object changes from pending to not pending, `object::alert()` must be called for the object. This function changes the state of all tasks "remembered" by the object from `IDLE` to `RUNNING` and puts them on the scheduler's `run_chain`. Thus all such operations should be member functions of the object's class or a related class. For example, in `qtail::put()`, if the queue was empty, a call to `alert()` is made for the associated queue head. If it was possible to put an object on a queue without calling a member function, then there would be no guarantee that `alert()` would be called.

The functions `task::waitvec()` and `task::waitlist()` suspend a task waiting for one of a list of objects, for example to wait for messages to arrive on one of a number of queue heads. `waitlist(object* ...)`<sup>9</sup> takes a list of object pointers terminated by a zero as argument, for example:

```
qhead*   q1;
qhead*   q2;
// ...
short    who = waitlist(q1, q2, 0);
```

will suspend the task executing it until either `q1` or `q2` is non-empty. If either is non-empty when `waitlist()` is called the task will continue immediately.

The value returned is the position in the list of the object that caused the return from the wait, that is if `q2` caused the task to resume the value 1 will be assigned to `who`. Positions are numbered starting from 0. `waitlist()` can take any number of arguments. The degenerate example

```
waitlist(0);
```

causes unconditional suspension of the task executing it without any guarantee of later resumption. It is equivalent to `sleep()` and `wait(0)`.

Please note that one should not assume that because `waitlist()` returns a particular value indicating one object as the cause of resumption none of the other objects are "ready". The value returned by `waitlist()` only indicates what is known to have happened, and it does not exclude other independent possibilities.

However if `waitlist()` indicates a particular object, that object is guaranteed to be "ready", because `waitlist()` does not return until the object is no longer pending.

---

9. `waitlist()` is an example of a function whose form does not satisfy current esthetic standards.

Because every class in the task system allows non-blocking examination of the conditions which might lead to suspension using the three wait functions, the value returned by `waitlist()` can always be ignored. The information it conveys can always be obtained by direct inquiry. In many cases, however, the value returned can be trusted and used to write simpler, more efficient programs.

`Waitvec()`, a variation of `waitlist()`, takes the address of a vector holding a list of object pointers. For example:

```
object*   vec[] = { q1, q2, 0 };
short     who = waitvec(vec);
```

is equivalent to the previous example.

### 5.1 System Time and Timers

The long variable `clock` measures simulated time. It is initialized to zero. It is illegal to assign to `clock`.

`Task::delay(int)` suspends a task for a specified time. That is,

```
long  t = clock;
delay(n);
actual_delay = clock-t;
```

will assign the value `n` to `actual_delay`. `Delay()` is useful for representing service delays in simulations. While a task is delayed in this way its state is still `RUNNING`, but it will not be affected by the actions of other tasks except if `cancel()` or `preempt()` is used on it. `Delay(n)` makes an `IDLE` task `RUNNING` so that it will start executing at time `clock+n`.

`Task::preempt()` makes a `RUNNING` task `IDLE` and returns the number of time units left of its delay. Applying `preempt()` to an `IDLE` or `TERMINATED` task causes a run time error. This function is useful when tasks are used to represent processes in a system with preemptive scheduling and delay times are used to represent the time used by executing processes. The value returned by `preempt()` allows the preempted task to be re-started with a new delay time which is a function of the delay time at the time of preemption. For example:

```
int  time_left = other_task->preempt();
// ...
other_task->delay(time_left+10);
```

A timer provides a facility for implementing time-outs and other time dependent phenomena.

Class `timer` has this declaration:

```
class timer : public sched {
public:
    timer(int);
    ~timer();
    void  reset(int);
    void  print(int, int =0);
};
```

A timer is quite similar to a task with constructor consisting of the single statement

```
delay(d);
```

that is, when a timer is created it simply waits for the number of time units given to it as its argument, and then wakes up any tasks waiting for it.

A timer's state can be either `RUNNING` or `TERMINATED`. This state can be inspected by using `sched::rdstate()`.

A common use of timers is to wait for a task and a timer. For example, one can wait for the completion of a task handling a simulated input operation and also on a timer. The timer ensures that the waiting task will eventually be resumed even if the input operation is never completed.<sup>10</sup>

```
timer*  tt = new timer(15);
short  res = waitlist(io_ptr,tt,0);
switch (res) {
case 0:
    /* normal completion of i/o */
    ...
    break;
case 1:
    /* time out occurred */
    ...
    break;
default:
    error(IMPOSSIBLE);
}
```

`Sched::result` and `sched::cancel()` have the same use and effects on timers as on tasks. Since there is no `timer::resultis()`, the value returned by `sched::result()` is undefined for a timer unless `cancel()` was used.

`Timer::reset()` re-sets the timer delay to the value of its argument. This makes repeated use of timers possible. A timer can be `reset()` even when it is `TERMINATED`.

A unit of simulated time can be used to represent any unit of real time. Only `delay()` causes the clock to advance.

## 6. MORE ABOUT QUEUES: CUTTING AND SPLICING

One of the most convenient and powerful ways of using tasks involves tasks defined to do a transformation on a data stream. Such a task is called a filter. It reads its input from one queue and writes its output onto another queue. Tasks at the "other ends" of these queues tend to view these queues plus the filter as one entity. The data source simply sees an output queue that is being emptied at some rate, and the task at the receiving end sees an input queue being filled. In other words, a task sees only its input and output queues and cares little about the "internal organization" of the programs that operate on the other ends of those queues.

For example, one task could produce a stream of lines of characters, that is objects of class `Line`, and another expect an input stream consisting of words, that is objects of class `Word`. A filter that handles the conversion could be defined and used like this:

10. In a quasi-parallel system this will only be true provided no infinite loop without task system calls exists. Such a loop constitutes an error that only a system with true parallelism or time slicing can recover from.

```

class Line_to_word : task
{
    Line_to_word(qhead*, qtail*);
    Word*   next_word(Line*);
};

Line_to_word::Line_to_word(qhead* in_q, qtail* out_q)
{
    Line*   l;
    Word*   w;
    for(;;) {
        l = in_q->get();
        while(w = next_word(l)) out_q->put(w);
    }
}

qhead*   line_q = new qhead(WMODE,10);
qhead*   word_q = new qtail(WMODE,50);
Producer* prod = new Producer(line_q->tail());
Consumer* cons = new Consumer(word_q->head());
Line_to_word* filt = new Line_to_word(line_q, word_q);

```

In this way the filter `filt` is programmed into the path between `cons` and `prod` using two queues to separate `filt`'s input from its output.

This is a fairly static use of a filter. Often one would like to insert a filter into an existing data path. For example, a macro-based text formatting program could be organized as a sequence of filters - each doing its small part of the common task. First some filters rearrange the input into a form suitable for the formatter proper, then the "input independent" formatter does its job producing output of a standard form, and last some output filters adjust this output to a form suitable for physical output. The task `filt` is an example of such a filter. In this scenario it would be useful to have each macro defined as a filter which the formatter proper inserts just in front of itself when the macro expansion is needed and which removes itself when it is not needed any more. Assuming that data streams are represented by queues, this can be achieved by using the class `qhead` functions `cut()` and `splice()`.

When the task `formatter` recognizes a call to the macro "foo" it creates a new task of class `Macro` to handle a macro of type `FOO` and diverts its own input through it. This is done by first "cutting" the input queue to create a place to insert the new filter, and then creating the filter giving it the new `qhead` and `qtail` as arguments:

```

qhead*   newhead = input_queue->cut();
qtail*   newtail = input_queue->tail();
Macro*   f = new Macro(FOO,newhead,newtail);

```

`Qhead::cut()` splits the queue to which it is applied into two. `Newhead`, the pointer returned from `cut()`, denotes the `qhead` for the original queue and has the same mode as the original `qhead`. The original `qhead` is now attached to a new empty queue with the same max as the original.

Puts to the original `qtail` will therefore place objects on the filter's input queue, and gets from the original `qhead` will retrieve objects from the filter's output queue.

The result of these operations has been to insert a filter with an input and an output queue into a queue without changing the appearance of that queue to anyone using it, and without halting the flow of objects through that queue. In our example the macro expansion filter `foo` will `get()` the input which would otherwise have gone to the formatter, interpret it as macro arguments, and output the expanded input as its output.

The filter can be removed again by splicing its input and output queues together with `qhead::splice()`:

```
newhead->splice(newtail);
```

`Splice()` deletes the `qhead` to which it is applied, the `qtail` given to it as an argument, and the queue denoted by that `qtail`. If the `splice()` operation causes an empty queue to become non-empty or a full queue to become non-full all tasks waiting for such a state change are resumed.

Deleting the filter completes the cleanup:

```
delete filt;
```

Typically a filter would remove itself when its task was completed, because the task that inserted it would not be programmed to be aware of the presence of the filter it inserted. The sequence of operations which enables a task to remove itself without a trace is:

```
cancel(any_value);
delete this;
```

This will work because `cancel()` does not imply immediate suspension, only a guarantee that the task cannot be resumed.

`Qtail::cut()` and `qtail::splice()` are similar to `qhead`, but they operate on the other end of the queue.

## 7. ENCAPSULATION

Passing information between tasks through queues can lead to rather tedious, repetitive (and therefore error prone) packing and unpacking of information into messages. Simple encapsulation techniques can be used to relieve the programmer of this. For example, by adding a constructor to the class `Message` the server example could be re-written thus:

```
class Message : object
{
public:
    int      r_operation;
    int      r_arg1;
    int      r_arg2;
    qtail*   r_reply;
    Message(int op, int a1, int a2, qtail* rp) :
        r_operation(op), r_arg1(a1),
        r_arg2(a2), r_reply(rp) {}
};

Message* mess;
rq->put(new Message(PLUS, 1, 2, reply_to));
mess = (Message *) rply->get();
if (mess->r_operation == ERROR) error();
```

Furthermore, because the message queues obviously are meant to hold only `Message` objects a specific message queue could be defined and used:

```
class Mqhead : qhead
{
public:
    Message* get() { return (Message *) qhead::get(); };
};
```

```

class Mqtail : qtail
{
public:
    int  put(Message* m) { return qtail::put(m); };
};

```

The use of `Mqtail::put()` ensures that only class `Message` objects are put on the queue, and no type cast is needed when class `Message` objects are taken from the queue using `Mqhead.get()`. For example:

```
mess = rply->get();
```

Because the body of `Mqtail::put()` is present in the class `Mqtail` declaration calls of `Mqtail::put()` will be expanded inline. This ensures that using a `Mqtail` is no less efficient than using a `qtail` directly. In many cases some error handling can also be handled by the derived `put()` and `get()` functions.

An alternative solution is to provide the server class with functions which handle the packing:

```

class Server : task
{
    qhead*  inp;
public:
    Server(char*, qhead*);
    int     plus(int, int, Mqtail *);
    int     minus(int, int, Mqtail *);
};

int Server::plus(int arg1, int arg2, Mqhead * rqt)
{
    Message* mess;
    int      x;
    inp->put(new Message(PLUS, arg1, arg2, rqt));
    mess = rqt->head()->get();
    x = mess->r_operation;
    delete mess;
    return x;
}

```

so now the server task can be requested to perform services like this:

```

Mqtail  qq;
Server  ss("plus_and_minus", 0, 0);
int     two = ss.plus(1, 1, &qq);
int     ten = ss.minus(12, 2, &qq);

```

For large programs this style of inter-task communication promises not only increased clarity, but also increased efficiency. The message queue interaction may, where necessary, be transparently replaced by a specially tailored inter-task communication facility.

These techniques are now widely applied in C++ programming, but when this paper was first written, they were new to C.

## 8. HISTOGRAMS AND RANDOM NUMBERS

To ease data gathering class histogram is provided.

```

struct histogram
// "nbin" bins covering the range [l:r[ uniformly
// nbin*binsize == r-l
{
    int    l, r;
    int    binsize;
    int    nbin;
    int*    h;
    long    sum;
    long    sqsum;
    histogram(int=16, int=0, int=16);
    void    add(int);
    void    print();
};

```

A histogram consists of `nbin` bins `h[0], ... h[nbin-1]` covering a range `[l:r[` of integers. The function `add()` adds one to the correct bin for its integer argument. The sum of the integers added is maintained in `sum`, and the sum of their squares is maintained in `sqsum`. If an argument to `add()` is outside the range `[l:r[` the range is adapted by either decreasing `l` or increasing `r`. The number of bins remains constant so the size of the range covered by a bin is doubled each time the size of the range `[l:r[` is. The `print()` function prints out the numbers of entries for each non-empty bin.

In most simulations some form of random number generation is needed. The generators provided here are intended to help the developer of a simulation to get started and to provide a paradigm for generators of more suitable distributions.

```

class randint
// uniform distribution in the interval [0,MAXINT_AS_FLOAT]
{
    long    randx;
public:
    randint(long s = 0);
    void    seed(long s);
    int     draw();
    float   fdraw();
};

```

The following program shows the use of class `randint`. The ints returned by `randint::draw()` are uniformly distributed in the interval `[0:largest_positive_int[`. The floats returned by `randint::fdraw()` are uniformly distributed in the interval `[0:1[`.

```

main()
{
    randint    ir;
    register   i;
    for (i=0; i<100; i++)
        printf("i=%d f=%f ", ir.draw(), ir.fdraw());
}

```

Each object of class `randint` provides an independent sequence of random numbers. `Randint::seed()` can be used to reinitialize a generator. The `draw()` function calls the underlying C library `rand(3)`. Using class `randint`, generators for other distributions are easily programmed. Note that `erand::draw()` calls `log()` from the math library, so a program using it must be loaded with `-lm`.

```

class urand : public randint
// uniform distribution in the interval [low,high]
{
public:
    int    low, high;
    urand(int l, int h) { low=l; high=h; }
    int    draw() { return int(low + (high-low) *
        (0+randint::draw()/MAXINT_AS_FLOAT)); }
};

class erand : public randint
// exponential distribution random number generator
{
public:
    int    mean;
    erand(int m) { mean=m; };
    int    draw();
};

```

## 9. IMPLEMENTATION DETAILS

The following sections contain many implementation-dependent details. The implementation described is the UNIX version. Implementation-dependent information is unfortunately often necessary to allow tuning and ease debugging.

This material is covered from a somewhat different point of view in reference 7.

### 9.1 Task Stack Allocation

The two arguments `mode` and `stacksize` allow the user to guide the system's handling of the task. Their exact interpretation is implementation dependent. Users who are not interested in implementation details and/or want a more portable program should set them both to zero. The system will then choose (hopefully reasonable) implementation-dependent default values.

The `stacksize` argument indicates the maximum amount of stack storage that the task is allowed to use. Using more is an error. It will be expressed in a unit of store (typically machine words) suitable for stack allocation on the host system.

The `mode` provides additional information. The value `SHARED` indicates that the stack space should be taken from the stack space of the parent task, that is the task which created the new task. Where `SHARED` stacks are used the active part of the stack is copied to a save area when a task is suspended, and copied back when it is resumed. *Since SHARED stack locations are not dedicated to a single task pointers to local variables should not be passed to other tasks.* The time needed to suspend and resume a task with `SHARED` stack is approximately proportional to the amount of stack space actually used at the time of suspension.

If, on the other hand, the `mode` is `DEDICATED` then a new and separate stack area is allocated, and no copying of stack space will occur.

### 9.2 Scheduling

Functions of a system class, known as the scheduler, are invoked as the result of any function of class `task` which causes the suspension of a running task, and may be invoked by any function from the standard classes described here. The scheduler selects the next task to run. When the scheduler finds no more tasks to run it examines the pointer variable `exit_fct`, and if this is non-zero the scheduler will call the function denoted by it.

Whenever `clock` is advanced the scheduler examines the pointer variable `clock_task`. If this denotes a task, then that task will be resumed before any other task. The `clock_task` must be `IDLE` when resumed by the scheduler. The class `task` function

`sleep()` is useful to ensure this.

### 9.3 Debugging and Tuning Aids

The task system has been designed under the assumption that a typical use of tasks may involve hundreds of tasks and need tuning to achieve an acceptable time-space tradeoff. The task of debugging such a system can safely be assumed to be non-trivial.

Classes were used in the implementation of the task system largely because they allow the scope of data and functions to be explicitly restricted to the object to which they belong. This allows better type checking of a multi-threaded program than could be achieved by a function-based implementation. The classes which constitute the task system were designed to allow quite strong type checking of programs using them.

A number of run time errors are detected by the task system. For example it is illegal to delete a queue on which a task is waiting. When such a run time error is detected the task system function `object::task_error` is called with the number of the error and the `this` pointer of the object which caused the error as arguments. Appendix B is a list of run time errors. `Task_error()` will in turn examine the pointer `error_fct`, and if this is non-zero call the function denoted by it with a copy of its own arguments. Otherwise `task_error()` will call the system function `exit()` with the error number as argument.

When returning from `task_error()` after executing an `error_fct` which returned rather than using `exit()` the task system will re-try the operation which caused the error (provided that `error_fct` could have affected the condition which caused the error). For example, a `put()` to a queue will be re-tried because the user's `error_fct` might have either caused the `get()` function to be used on the queue, or used `chmax()` to allow more objects to be inserted into that queue.

This error handling mechanism is primarily designed for debugging and it is expected that user error functions will print some appropriate error message and exit.

Beware of infinite loops.

All task system classes have a function `print()` which can be used to print the contents of their objects on `stdout`. A `print()` function takes an `int` argument indicating the amount of information to be printed. `Print(0)` gives the minimum amount of information, `print(VERBOSE)` rather more, and `print(CHAIN)` will call `print()` for objects on lists associated with the object with its own arguments. The `print()` argument constants can be combined by the `or` operator. For example

```
thistask->print(VERBOSE);
run_chain->print(VERBOSE|CHAIN);
```

will verbosely describe every non-TERMINATED timer and every RUNNING task. For tasks information about the run time stack is printed by `print(STACK)`. If the function `hwm()` has been called `print(STACK)` will also give an estimate of the maximum amount of stack space ever used by the task, the stack's "high water mark". For tasks that share a stack, the high water mark printed will be the high water mark of most greedy task. For example, information describing stack usage for all tasks can be printed by:

```
task_chain->print(STACK|CHAIN);
```

The output of the `print()` functions is implementation-dependent and hopefully self-explanatory.

### 9.4 Overheads and Performance

The store used for representing a class object in addition to the user specified data is:

object	3 words
timer	5 words
task	18 words + stacksize
queue	15 words (including the qhead and the qtail)

The times needed to execute some of the task system functions are (very) approximately:

C procedure call + return	1 unit
task suspend + resume	9 units (using result())
put	2 units
get	2 units
wait, waitvec, or waitlist	3 units

The last four actions can all cause a task to be suspended. When this happens add 6 units of time.

For timing results relative to UNIX process switching, see reference 7.

The task system uses about 15K bytes of store for program and data, but much of this is redundant virtual function tables that will be eliminated in a future version of the C++ compiler.

#### 10. ACKNOWLEDGEMENTS

The task system is in many ways a descendant of A. G. Fraser's set of C functions. M. D. McIlroy acted as "midwife" for many parts of the design.

## REFERENCES

1. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.
2. Brinch-Hansen, Per. *The Architecture of Concurrent Programs*. Prentice Hall, 1977
3. Butler W. Lampson and David D. Redell. "Experience with Processes and Monitors in Mesa." *Comm. ACM* 23(2):105-117, February 1980.
4. Martin Richards and Colin Whitby-Stevens. *BCPL: The language and its compiler*. Cambridge University Press, 1979.
5. A. G. Fraser. "C Language Routines for Multi-Thread Computation." Bell Telephone Laboratories Internal Memorandum 1979.
6. G. M. Birtwistle et al. *SIMULA begin*. Auerbach, 1973.
7. Jonathan E. Shopiro. "Extending the C++ Task System for Real-Time Control". this volume.

## Objects

The task system as described above is implemented using a lower level of abstraction based on the direct use of the class `object`. Class `object` can also be used as a base for other (user defined) abstractions, but beware, it is an implementation tool that is not intended to be used directly.

Class `object` is a base class for all classes in the task system and also the most basic facility for inter-task communication. The declaration of class `object` looks like this:

```
class object
{
    friend sched;
    friend task;
    olink*      o_link;
public:
    object*      o_next;
    virtual int  o_type();
    object() { o_link=0; o_next=0; }
    ~object();

    void         remember(task* t) { o_link = new olink(t,o_link); }
    void         forget(task*);    // remove all occurrences of task from chain
    void         alert();          // prepare IDLE tasks for scheduling
    virtual int  pending();        // TRUE if this object should be waited for
    virtual void print(int, int =0); // first arg VERBOSE, CHAIN, or STACK
};
```

The task system implements objects of type `TASK`, `QHEAD`, `QTAIL`, and `TIMER`.

Virtual functions make it unnecessary to ever test the type of an object. The virtual function `o_type()` is never called.

A task can be added to the set of tasks "remembered" by an object by executing `object::remember()` and a task can be removed from this set by executing `object::forget()`. Executing `object::alert()` has the effect of transferring all IDLE tasks remembered by the object to the `run_chain` and the `RUNNING` state.

The virtual function `object::pending()` provides the "glue" that allows new kinds of objects and new communication protocols to be added to the task system. The object may have any kind of operation that may cause the invoking task to wait, but it must implement its own version of `pending()` to tell whether the operation would cause a wait.

A task can be "remembered" by several objects or several times by the same object without any ill effects. `Forget()` will insure that its argument is not "remembered" any more, and it causes no bad effects when used for an object that does not "remember" its argument task. No record is kept of how many `alert()` operations have been executed on an object. `Alert()` does not cause an object to forget() tasks. Executing a `remember()` does not suspend a task. Applying `alert()` to an object that does not remember any tasks is legal, but has no effect. *Caveat emptor!*

The functions `object::remember()`, `object::forget()`, `object::pending()`, and `object::alert()` provide a simple, efficient, but unstructured and therefore error-prone, communication mechanism.

The declarations for the task system classes can be found in `/usr/include/CC/task.h` on systems where it is implemented.

## Run Time Errors

When an error is detected at run time, `task_error()` is called. This function will examine `error_fct` and if this variable denotes a function, that function will be called with the error number and this as arguments, otherwise the error number will be given as argument to `print_error()` which will print an error message on `stderr` and terminate the program.

[E_OLINK]	Attempt to delete object which remembers a task.
[E_ONEXT]	Attempt to delete an object which is still on some chain.
[E_PUTFULL]	Attempt to put to a full queue.
[E_PUTOBJ]	Attempt to put an object already on some queue.
[E_BACKOBJ]	Attempt to putback an object already on some queue.
[E_BACKFULL]	Attempt to putback to a full queue.
[E_GETEMPTY]	Attempt to get from an empty queue.
[E_SETCLOCK]	Clock was non-zero when <code>setclock()</code> was called.
[E_RESTERM]	Attempt to resume TERMINATED task.
[E_RESRUN]	Attempt to resume RUNNING task.
[E_NEGTIME]	Negative argument to <code>delay()</code> .
[E_RESOBJ]	Attempt to resume task or timer already on some queue.
[E_HISTO]	Bad arguments for histogram constructor.
[E_TASKMODE]	Bad mode for task constructor.
[E_TASKDEL]	Attempt to delete non-TERMINATED task.
[E_TASKPRE]	Attempt to preempt a non-RUNNING task.
[E_TIMERDEL]	Attempt to delete a non-TERMINATED timer.
[E_SCHTIME]	Scheduler chain corrupted: bad time.
[E_SCHOBJ]	Scheduler chain corrupted: bad object.
[E_QDEL]	Attempt to delete a non-empty queue.
[E_CLOCKIDLE]	The clock_task was non-IDLE when clock was advanced.
[E_STACK]	Task run time stack overflow.
[E_STORE]	No more free store - new failed.
[E_RESULT]	A task attempted to obtain its own result().
[E_WAIT]	A task attempted to wait() for itself to TERMINATE.
[E_FUNCS]	FrameLayout - func start.
[E_FRAMES]	FrameLayout - frame size.
[E_REGMASK]	FrameLayout - register mask.
[E_FUDGE]	fudge return - no place to copy.
[E_NO_HNDLR]	sigFunc - no handler for signal.
[E_BADSIG]	illegal signal number.
[E_LOSTHNDLR]	signal handler not on chain.

## A Program using Tasks

```
#include <class_task.h>

/* trivial test example:
   make a set of tasks which pass an object round between themselves
   use printf to indicate progress
   WARNING: this program sets up an infinite loop
*/

class pc : task
{
    pc(char*, qtail*, qhead *);
};

pc::pc(char* n, qtail* t, qhead * h) : (n,0,0)
{
    printf("new pc(%s,%d,%d)\n",n,t,h);

    while (1) {
        object* p = h->get();
        printf("task %s\n",n);
        t->put(p);
    }
}

main()
{
    qhead* hh = new qhead;
    qtail* t = hh->tail();
    qhead* h;
    short i;

    printf("main\n");

    for (i=0; i<20; i++) {
        char* n = new char[2]; /* make a one letter task name */
        n[0] = 'a'+i;
        n[1] = 0;

        h = new qhead;
        new pc(n,t,h);
        printf("main()'s loop\n");
        t = h->tail();
    }

    new pc("first pc",t,hh);
    printf("main: here we go\n");
    t->put(new object);
    printf("main: exit\n");
    thistask->resultis(0);
}
```

## C++ versus Lisp: A Case Study

Howard Trickey  
AT&T Bell Laboratories, Room 2C460  
Murray Hill, New Jersey 07974

### Abstract

A large application program was simultaneously coded in both C++ and Lisp. This note compares the two implementations. The conclusion is that programming in the two languages takes roughly the same amount of time and effort, but C++ seems better for a number of reasons.

### Introduction

How can one compare one programming language with another? Faced with the prospect of doing a large program, and unhappy with the language I had used on a previous project (sugared Pascal), I wanted to compare two likely candidates: C++ [5] and Common Lisp [4]. I decided to do a smaller but still substantial application in both languages, to try to gauge how the languages affected the program and the program development process.

### The Application

The application chosen for the case study was a graph drawing program called *drag*. It reads a graph description—information about nodes and edges—and decides how to lay the graph out on paper so that it looks good. *Drag* is a “typical” program: it scans and parses, uses a number of complicated data structures (e.g., to keep track of topology), has combinatorial algorithms (e.g., shortest path finding), and numerical algorithms (a linear program solver for balancing aspects of aesthetic node placement).

There were no specifications for *drag*: design and coding went on together, and I was the only person involved. This means that my comments are most directly relevant to situations involving one person doing research prototypes. However, *drag* may eventually become a widely used tool, so the maintenance and documentation aspects of the code are important too.

At first, about half of *drag* was done only in Lisp, and then the C++ version was coded by essentially transliterating the Lisp version. After that, I alternated between which version I changed first. *Drag* is now functionally complete, with identically behaving C++ and Lisp versions. The result of the comparison is that I will abandon the Lisp version, and just use C++.

It is quite possible that someone with a different background and ability would come up with a different result, so you should know a bit about me to help interpret this note. I've been programming for fifteen years in a variety of programming languages, including C, PL/I, AlgolW, Pascal, Lisp, Snobol, APL, and Fortran. Table 1 shows my biggest programs before this experiment. You can see that I have much more experience in languages related to C++ than in languages related to Common Lisp. However, I've done a number of smaller Lisp programs, and feel I have a reasonable facility in the language. I'd had no previous experience with C++ or any other language that supports object-oriented programming.

### Statistics

Table 2 compares the C++ and Lisp versions of *drag*. The statistics reflect *drag* version 22 May '87, AT&T's C++ version cfront 8 August '86, and Lucid's Sun Common Lisp 2.0.3 December 19 '86

Program	Lines	Language
silicon compiler	28,000	WEB (sugared Pascal)
parser generator	8,000	C
production compiler	4,000	PDP/11 assembly language
graph drawer (another)	3,000	Franz Lisp
commercial database updater	2,000	PL/I
student operating system	2,000	PL/I
student compiler	2,000	PL/I
regular expression compiler	1,000	C

Table 1: Author's previous large programs

		C++	Lisp
Source code	lines	11,123	9,385
	tokens	36,513	38,406
	chars	287,304	278,231
Object code	kbytes	279	360
	env. kbytes	471	5707
Compile time	cpu min:sec	12:10	21:53
	real min:sec	15:55	27:37
Average Execution time	cpu sec.	7.1	24.1
Big Test Execution time	cpu sec.	109.4	805.5
Debug cycle	elapsed sec.	150	10

Table 2: Comparing *drag* implementations

[6]. The hardware is a diskless Sun 3/75 with 16Mbytes of memory, connected over a lightly loaded ethernet to a Sun 3/180 with Eagle disks and 12Mbytes of memory.

#### Source code size

The source code size is about the same in the two versions— C++ is bigger by some measures, Lisp by other ones. This result is unexpected, perhaps. Lisp has many built-in datatypes and operations lacking in C++, and some powerful Lisp control forms (e.g., mapping) are hard to duplicate in C++ because of its tight type checking. *Drag*'s basic data structure definitions and operations take 1400 lines of C++ versus 500 lines of Lisp, so there is indeed an advantage to Lisp's built-ins, an advantage that might be telling in a smaller application.

The supposed control structure advantage for Lisp didn't materialize, mainly because the data structure definitions in the C++ version were carefully designed to define appropriate access and iteration aids. For example, the Lisp code to call a function `fixnodetype` on each of a list of `nodetypes` looks like

```
(map 'nil #'fixnodetype (graph-nodetypes *g*))
```

and it looks like this in the C++ version:

```
nodetypeploop(nty, g.nodetypes)
    fixnodetype(nty);
```

(`nodetypeploop` is a macro for iterating over a set of pointers to `nodetype`; it declares `nty` to be a `nodetypep`.) The following code makes heavy use of Lisp machinery

```
(reduce #'union (mapa 'list #'valgaps (nodeperms n)
                      ue (node-first-edge n)))
```

(`mapa` is my macro which expands to a mapping over one list, supplying the same extra arguments to each call.) The C++ version had to rewrite `val-gaps` as a function that adds to a supplied set, but `addvalgaps` is no longer or harder to understand.

```
edgepset* ans=new edgepset;
permploop(p, perms(n))
    addvalgaps(ans, p, ue, first_edge(n));
```

Of course one reason why the two versions are about the same size is that my desire to have identically behaving programs gave a strong incentive to look at one version when coding the other. Perhaps a True Lisper has a completely different, more compact coding style. I did notice a difference in the "natural" way to express things in one language versus the other. One of the main differences is that you are more aware of storage management in C++, so there is a tendency to use a procedure that mutates a global or a passed argument instead of creating and returning a new list as is natural in Lisp. Another difference is that Lisp lore strongly encourages recursion in places where a C++ user would iterate. I tried to allow these differences to exist in the two versions, though I probably underused recursion in Lisp. The differences don't seem to affect size very much. I don't think there are any places where one language would have dictated a vastly different design than the other. (But see the *Execution time* section, below, for an anecdote about one case where the two versions departed significantly.)

It probably took longer to type the Lisp *drag*, but not significantly in comparison to think time.

#### Object Code size

The C++ executable is 750 kbytes when compiled for debugging; 471 kbytes of that can be removed by stripping the symbol table. The Lisp object files take up 360 kbytes, and they must be loaded in on top of the Lisp image, an additional 5707 kbytes. So the C++ files take up more disk space, but it takes much more memory to run the Lisp *drag*. I found that 8 Mbytes of real memory does

not yield adequate performance, even on a Sun with a disk. To run the Lisp compiler and the Sun window system at the same time on a diskless Sun, 16 Mbytes is adequate and seems to be necessary (12 Mbytes might also work). Another problem is swap space: we configured our diskless nodes with 20 Mbytes of swap space, and that isn't nearly enough. It's not easy to increase swap space, nor is it desirable with our limited number of disks.

### *Compile time*

Cfront is not a speed demon, but the Sun Common Lisp compiler is quite slow: compiling the whole system takes about twice as much time for Lisp compared to C++. The 2.0.3 version of Sun Common Lisp has much better compiler performance than the 1.2 version that I used during most of the development. The earlier version was about four times slower than Cfront, so that recompiling a 500 line Lisp file took five or six minutes versus one minute for the equivalent C++ file. Those figures were on opposite sides of my dividing line between "acceptable" and "annoyingly long". Lisp compiles took such a long time that I tended to avoid doing them very often. But that had problems of its own: certain common coding errors are best discovered by compiling (e.g., it warns about misspelled variables "assumed special"). The new Lisp compiler takes about the same time as the C++ compiler for some source files—it is only garbage collections and one very long procedure (a scanner) that cause the system build time to be twice as long. So my experiences might have been different had I had the new compiler all along.

However, for finding syntax errors, C++ has an big advantage. If a 500 line C++ file has errors, you will find out about all of them (up to a limit) in about 20 seconds. The Lisp compilation aborts after the first error, so it can take many minutes to find them all.

### *Execution time*

Drag's average execution time on a sample of 25 graphs was 7.1 and 24.1 seconds for the C++ and Lisp versions, respectively. The Lisp version was compiled, not interpreted, and all of those test cases ran without garbage collection. A rather bigger graph took 109 seconds in C++ and 805 seconds in Lisp, because Lisp needed 16 garbage collections to finish. The compiler flags for both versions were as I set them for normal development: `-g` for C++ and the defaults for Lisp. The C++ version could be sped up using optimization (`-O`), and the Lisp version could be sped up using `fast-entry` (avoids checks for the correct number of arguments).

Both versions were slightly tuned. There is an interesting story here. When only the Lisp version existed, the average execution time was on the order of minutes (for only part of the functionality). Sun Common Lisp is hard to tune: the only way seems to be to put calls to prints of (`get-internal-run-time`) at strategic points in the program, refining "strategic points" as the bottleneck is isolated. Because this is so tedious, I hadn't done it, and assumed that my algorithms were inherently inefficient—to be fixed later. Then I got the C++ version working and found that it ran 15 to 20 times faster. That seemed a little excessive, so I isolated the Lisp bottleneck. It turned out to be a shortest-paths routine—the only place where the two versions differed significantly. The Lisp shortest-paths function was very general and elegant: it took a list of objects and a function that could be called on two objects to yield the distance between them, and returned two functions (closures) that could be called on two objects to yield the length of the shortest path and a list of objects on that path. That was hard to express in C++, due to type checking problems (I needed to call this function with several different types of objects), so the C++ version took an adjacency matrix and set up global data structures that could be examined to get the desired results. It required significantly more code in the calling routines, but ran much faster. I changed the Lisp version to use the same technique as the C++ one, and Lisp sped up to numbers close to those in Table 2.

After fixing the Lisp shortest-paths function, and after more development, the bottleneck became a linear program solver. I spent an afternoon trying to discover what type declarations to add to the Lisp version to get the best code out of the compiler (there is a `disassemble` function to help with this). It turns out that giving the most detailed information about the type of certain floating vectors was not as good as giving less information! It would take a lot of experience to know exactly what style

	Compile Time (min.)		Debug Time (min.)	
	C++	Lisp	C++	Lisp
1 (C++ first)	31	30	42	54
2 (Lisp first)	28	34	1	4
3 (C++ first)	6	16	7	15
4 (Lisp first)	3	7	19	36
total	68	87	69	109

Table 3: Four modifications to both versions

Lisp should be written in to get the fastest code.

As far as the C++ version goes, it was quite easy to use the *gprof* tool [2] to profile the code and discover that the biggest fixable bottleneck was the allocation of a certain type of structure. Replacing it with a class-specific allocator reduced the runtime by 10%, and no further tuning was done (the bottleneck is the linear program solver, just as in the Lisp version).

There are several morals to this story. The first is that Lisp is hard to tune: one can easily be unaware that a simple change would give immense improvements; or, something that seems like it should help (adding more detailed type information) can sometimes hurt. And second, in cases where Lisp features can be used to do things that are hard to do in C++, you have to beware of performance penalties.

#### Debug time

The "debug cycle" figures in the table refer to the amount of time between making a source file change and testing it. Here Lisp has a big advantage, because you can interpret the source file currently being debugged, and it only takes about 10 seconds to reload a source file. With C++, the source file must be compiled and then all the object files must be relinked (a not inconsiderable job—there are currently 1700 external symbols in *drag*), for a total of about 150 seconds. In this situation the Lisp performance is good, whereas the C++ performance is "annoyingly long". Sun is apparently going to have incremental loading in its debugger in some future release; that will improve things somewhat.

On the other hand, once debugging starts I prefer C++ debugging (using *dbx* [1]) to debugging Lisp with Sun Common Lisp's current breakpoint handler and stepper. The most annoying problem in the Lisp environment is the need to march up and down the stack to where a local variable is bound before you can inspect it. And when stepping, there are a lot of irrelevant binding forms to step over. Even better than *dbx* for debugging C++ is *dbxtool*, which continually shows the executing line during stepping, and lets you print a variable's value by pointing at it with a mouse. I imagine the Lisp environment will improve, because it is close to unacceptable right now.

Table 3 represents an attempt to compare elapsed times when identical changes were made to the C++ and Lisp versions. On four separate occasions I set out to make a specific change to *drag*, as follows:

1. Code one version.
2. Code the other version, looking at the first.
3. Use the compilers to find and remove all the errors that the compilers can find, trying to pay no attention to errors found in one version while correcting the other.
4. Debug the first-coded version by testing. For C++, this involves multiple compiles; interpret the Lisp until debugging seems finished, then compile it (but occasionally I also ran the Lisp compiler after a particularly heavy modification).

## 5. Debug the other version, trying to pay no attention to errors found while debugging the other.

The C++ version was coded first in tests 1 and 3, and the Lisp version was coded first in tests 2 and 4. The order didn't really matter much for these tests, because many of the bugs were superficial coding bugs that were completely different in the two versions. I typed commands during the tests to record when I started and stopped compiling and debugging. "Debugging" means running the program on test data, setting breakpoints and occasionally stepping, inspecting variables, and thinking about where errors could be, but not the time to think about how to change the source code to fix the error or the time editing in those changes. The "time compiling" figures are probably less significant than the "time debugging" figures, especially when I tell you that the relatively small task size meant that I ran the Lisp compiler rather more frequently than was typical overall.

There isn't really enough data in Table 3 to make strong conclusions about the relative development time in C++ versus Lisp.<sup>1</sup> Really, I should have tried doing the same modification task in each language several times, to estimate the variance. I suspect the variance is rather high, because a moment's lack of attention can lead to an error that makes the difference between needing another compile or not. Also, more and longer tasks should have been tried. The reason they weren't is just that doing these experiments is very tedious and anxiety-provoking—nobody likes to do the same thing twice. And, as the tasks get longer, the coding and debugging process becomes more entwined with algorithm design, and it soon becomes impossible to remember and reproduce the blind alleys one follows in the program's design.

As a result of my experiments, I believe that developing a large program in Lisp is not significantly faster than doing it in C++, and there is some evidence that doing it in C++ is modestly faster in my current environment.

### Qualitative Aspects

There are other aspects that go into choosing a programming language besides those that can be characterized with numbers. They affect reliability, maintainability, and general ease of programming.

### Syntax

To me, a C++ function is easier to read than the equivalent Lisp function. Assignments, function calls, and control structures all look rather different from each other in C++, whereas they all look about the same in Lisp. This is important if you want to skim a function to get an impression of what it does, or to look for some specific feature.

One of the best things about C++ syntax compared to Lisp syntax is the way that new local variables are declared and bound. A C++ statement like

```
edge* esym=sym(e);
```

can be used almost anywhere in a function, causing a minimum interruption, while the Lisp equivalent

```
(let ((esym (edge-sym e)))
  ... ; use local esym here
)
```

is more complicated and causes another level of indentation. The desire to avoid excessive indentation distorts my Lisp code away from what would otherwise be better code. You can gather together several local bindings in a `let`, so sometimes I succumb to the temptation to declare things before they can be bound. That means they can't be declared to have the type that they will eventually have, so the compiled code will be poorer. If you bind new Lisp locals where logical, a function that is easily comprehensible in C++ looks too long in Lisp. I find coding Lisp more tiresome, because of the need for constant binding planning; and changing the code requires more editing.

<sup>1</sup>Under some assumptions, the ratio of Lisp debugging time to C++ debugging time is significantly greater than one (at a 5% level), while the compile time ratio is not significantly greater than 1.

```

(defvar *c* nil)
(proclaim '(type list *c*))

(defun finc (x)
  (declare (type float x))
  (+ x 1.0)
)

(defun try ()
  (finc *c*)
)

```

Figure 1: Uncaught type error in Sun Common Lisp

Perhaps a Lisp advocate would counter that it isn't difficult to customize the Lisp syntax to whatever one desires. My experience is that this isn't a good idea, because tools like the debugger operate on a more canonical representation so it is best to be familiar with your program in "standard" Common Lisp. It is also to the benefit of other readers not to use a strange syntax.

Common Lisp allows some optional function parameters to be specified by "keyword". This occasionally makes for code that is easier to understand at a glance.

### *Type Checking*

C++ does extensive type checking at compile time. Sun Common Lisp checks for inconsistencies in the number of arguments supplied to a given function, but that seems to be about all the type checking it does at compile time. For example, the only way to detect that the code in Figure 1 has a type problem—`try` calls `finc` with a list instead of a float—is to exercise a test case that calls `try`. Even more insidious are errors that aren't caught at run time, but nevertheless are type errors that may come back to haunt you. If the type of `*c*` in Figure 1 is changed to `fixnum`, the compiled code currently runs fine (because the compiler makes no use of the declaration in `finc`), but the compiler might change some day, and cause this code to bomb.

It takes longer to debug code if the errors have to be discovered by exercising test cases instead of being pointed out by the compiler as a batch of type errors.

Incidentally, the type checking facilities of C++ exceed those of the C and lint combination [3], and this is enough reason in itself to use C++ instead of C.

### *Primitives*

Common Lisp has over 700 built-in functions, macros, special forms, variables, and constants. This is definitely an asset: when you have to hand-tailor a C++ function instead of calling a built-in Lisp function, it takes time to write and debug, and chances are that it won't be as well coded as the polished Lisp primitive. And one can imagine the Lisp compiler using special knowledge of its primitives to make optimizations that would be difficult to discover in C++. It took me about two weeks to design and implement C++ things that were primitives in Lisp; after that, the added repertoire of Lisp didn't make much of a difference.

Another advantage of having a wide selection of built-in functions is that careful thought has gone into their definition, so that you get a more rational and easy-to-remember suite than an ad hoc set of utilities made for a specific C++ program. An outsider coming to *drag* for the first time would have to spend some time studying my C++ definitions of sets, sequences, and tables before looking at the rest of the code, but the Lisp version of *drag* uses sequence and hash table functions that will be familiar to any Common Lisp user. C++ will eventually get a more complete library, but Lisp is far ahead at this point. It isn't clear that C++ can ever have as nice a library as Lisp: for one thing, you

might need a C++ “universal class” with a very large number of virtual functions and that doesn’t seem to be a great solution. C++ will probably support parameterized types some day, and that might be sufficient to build a good library.

A disadvantage of Lisp’s built-in primitives is that I overused lists. Often I used lists to implement “sequences” that would be built incrementally by adding new elements to the end: with lists, you push the new element onto the beginning and reverse the list after the sequence is completely built. It’s possible to forget the reverse, and the implementation isn’t as good as one that keeps a separate end-of-list pointer. Making a proper append-at-end sequence abstraction isn’t hard to do in Lisp, but I didn’t do it because it means forgoing the nice primitives that work on lists. In retrospect, it would have been a good idea. Similarly, there were other occasions where the path of least resistance was to use lists of lists for certain data structures, and some hard-to-find bugs derived from creating and using lists that didn’t have the assumed structure. Making the special classes in C++ seemed annoying at the time, but it paid off in better code.

### Memory Management

I had expected that the automatic garbage collection provided by Lisp would ease the programming burden considerably, because many of the hardest bugs to find in my previous large programs were due to improper freeing of memory. It turned out not to be as much of an advantage as expected. First, you can afford to be more profligate with memory in a C++ program because you don’t have to share memory with a huge Lisp image. And second, C++ classes provide a reasonable mechanism to handle much of the allocating and freeing automatically. I tend to write in a slightly different style in C++ to avoid massive garbage buildup—e.g., fill a global set instead of returning a new one—and the result isn’t as clean as the Lisp version. But the payback is runtime efficiency.

### Testing and Debugging

Sitting in a Lisp interpreter offers the possibility for better testing and debugging:

- If you want to test some small function, you simply call it with appropriate arguments and see what it does.
- If the program bombs in some strange way, you can enter the Lisp “inspector” to wander around in the data structures and discover problems.
- When you discover and fix a bug, only the changed function need be reloaded.

The first point turned out to be quite unimportant in *drag*’s development. The problem is that *drag*’s internal data structures form a complex intertwined web, and a large fraction of the functions access those data structures (directly or indirectly). As a result, the easiest way of testing a function was to let it be called in the context of a run of *drag* as a whole—the same way the C++ functions were tested.

The Lisp inspector is sometimes a nice way to look at *drag*’s data structures. To get the equivalent effect in C++, I had to write a number of *print* functions to be called from the debugger. An initial cost—but then it was almost as good as the Lisp inspector.

Lisp’s incremental loading was the feature I most wanted in C++. However, my experiments have shown that my total C++ debugging environment is apparently superior to my total Lisp debugging environment in terms of total time spent, and that’s the bottom line.

### Other

Some readers may wonder about the “object oriented” facilities of C++, and why I didn’t compare to a Common Lisp enhanced with similar features. I used C++ classes to abstract my data structures, to force access to them through a few selected functions, and to make sure they are initialized reasonably. Lisp struct definitions give some of these effects, and its package concept helps with hiding. I made very little use of C++ derived classes and virtual functions. Sun Common Lisp release 2.0 has *flavors*,

		Kyoto Lisp
Object code	kbytes	837
	env. kbytes	1859
Compile time	cpu min:sec	20:37
	real min:sec	23:14
Execution time	avg. cpu sec.	31.4
Debug cycle	elapsed sec.	10

Table 4: Kyoto Common Lisp statistics

providing an object-oriented interface, but *drag* was programmed before that release was available. It appears that using *flavors* might incur quite a performance degradation, so I probably wouldn't have used it anyway.

Another aspect affecting the choice of language is the portability of the resulting program. Here again C++ wins, because any Unix operating system has C, and it isn't difficult or expensive to get C++ running on top of that. And C++ is already in use under other operating systems, including MS-DOS, DEC-VMS, and IBM-VM. Common Lisp is much less common, and implementations are expensive enough that there has to be a good reason to install it.

#### Another Common Lisp

Another implementation of Common Lisp is Kyoto Common Lisp [7]. I had a chance to try it after the comparison experiment was over. Table 4 shows those statistics from Table 2 that changed. Compilation is about the same as with Sun Common Lisp, but the average execution time on my *drag* tests is 30% longer with Kyoto Common Lisp.

The KCL compiler does some type checking, warning about some cases where declarations conflict (but it still misses the error in Figure 1). The breakpoint handler allows more convenient access to the local variables than the current Sun Common Lisp handler.

#### Conclusions

There are some advantages to programming in Common Lisp, but more disadvantages, so from now on I will only maintain the C++ version of *drag*, and I would choose C++ over Common Lisp for another large program. Some of the problems I found with Lisp were specific to the current Sun Common Lisp, which will no doubt be improved. Specialized Lisp workstations can make up the performance difference, at the cost of more expensive hardware, and they have a more integrated and fully developed user interface, so that debugging time might have been shorter had I used one of them. But the biggest difference between C++ and Lisp—type checking—will always remain; I like the security and maintainability that type checking gives. Furthermore, as Common Lisp implementations improve, so will C++ implementations. It seems likely that a better and faster C++ compiler will appear and that some form of incremental compilation scheme will eventually be used.

Perhaps the most significant result is that there does not appear to be a substantial difference in the time to develop a program in C++ versus Common Lisp, at least for large enough programs.

#### References

- [1] Evan Adams and Steven S. Muchnick. Dbxtool: a window-based symbolic debugger for sun workstations. *Software—Practice and Experience*, 16(7):653–669, July 1986.

- [2] Susan L. Graham, Peter B. Kessler, and Marshall Kirk McKusick. An execution profiler for modular programs. *Software—Practice and Experience*, 13(8):671–685, August 1983.
- [3] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1978.
- [4] Guy L. Steele Jr. *Common LISP: The Language*. Digital Press, Burlington, MA, 1984.
- [5] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.
- [6] *Common Lisp User's Guide*. Sun Microsystems, Inc., under license from Lucid, Inc., Mountain View, CA, 1985.
- [7] Taiichi Yuasa and Masami Hagiya. *Kyoto Common Lisp Report*. Ibuki, Los Altos, CA, 1985.

1. The first step is to create a new directory for the project.  
2. Next, we need to create a new file named `main.cpp`.  
3. We will use the `g++` compiler to compile the code.  
4. The output of the compilation will be an executable file.  
5. Finally, we will run the executable file to see the results.

## Avalon/C++:

# C++ Extensions for Transaction-Based Programming

David Detlefs, Maurice Herlihy, Karen Kietzke, Jeannette Wing<sup>1</sup>

### Abstract

Avalon extends C++ to support transaction-based programming. Avalon allows programmers to implement classes of *atomic objects*, objects that provide *failure atomicity*, *permanence*, and ensure the *serializability* of the transactions that perform operations on them. Classes gain these properties by inheritance from a set of built-in classes. Atomic objects are encapsulated in processes called *servers*; Avalon provides syntactic constructs for defining and executing servers. Constructs are also provided for beginning, ending, and aborting transactions, and for executing transactions concurrently.

Avalon is being implemented as a preprocessor which translates Avalon code into C++. This preprocessor is being built by modifying the C++ preprocessor. The code we generate makes extensive use of the Camelot transaction processing system [9], which, in turn, relies on the Mach operating system [1].

## 1. Introduction

This paper has four parts. First, we motivate the need for Avalon, as language support for programming reliable distributed systems (section 2), and highlight its key features (section 3). Avalon is interesting in its own right as an example of the extension of object-oriented programming to a new application domain. Second, we describe our experience extending C++ (section 4), which may be of interest to others considering extending C++ for other uses. We briefly give some comments on our experience in using C++ (section 5), and close with an outline of our future plans (section 6).

---

<sup>1</sup>This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract F33615-84-K-1520. Additional support for J. Wing was provided in part by the National Science Foundation under grant CCR-8620027.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

## 2. Motivations for Avalon

Concurrency and fault-tolerance are the two key properties of large software systems that Avalon addresses. The object-oriented programming paradigm allows concurrency to be easily introduced into systems. For example, the “message-passing” metaphor used by systems such as Smalltalk corresponds well to the communication techniques used by concurrent languages like CSP [7]. However, few object-oriented languages actually support concurrency directly. Some significant exceptions include Orient84K [8], Concurrent Smalltalk [10], and Actors [2]. We believe that it is important to realize that adding concurrency to an object-oriented language must be more than simply adding a “cobegin” statement and locking primitives. In an object-oriented language, concurrency control can and should reside in the objects. Avalon provides facilities that allow objects to synchronize processes that are attempting to execute operations on them concurrently. Avalon supports concurrency both at the level of multiple processors on a single machine and at the level of multiple machines in a network.

We address fault-tolerance, particularly in conjunction with concurrency, by relying on known serialization and recovery techniques from the database community. Databases themselves are logical candidates for being recast in the object-oriented paradigm. The view is that databases are objects supporting various “lookup” and “insert” operations, and that the records in the database are themselves other objects of the appropriate record type. Database systems, however, often have very stringent reliability requirements, which are commonly satisfied by grouping operations on the database into sequences called *transactions*. Modifications made by a transaction become permanent if and only if the transaction completes; if something goes wrong (the transaction *aborts*) the modifications are undone. Avalon provides transaction semantics through the objects that transactions manipulate. These *atomic objects* provide for their own recovery in the event of a transaction abort. Avalon provides a set of built-in atomic classes, as well as facilities that allow users to define new ones.

Taken together, network- and processor-level concurrency and database-style fault-tolerance form an application domain called *reliable distributed computing*. This domain is already important; systems such as automated teller machine networks and airline reservation systems are real-world examples. We believe that many more systems will be built this way in the future, especially if it becomes easier to implement them. A high-level programming language such as Avalon, which supports this domain, will greatly reduce implementation complexity. Providing features in high-level constructs will avoid errors, by eliminating the need for hand-crafting complicated operating system-level code for each instance of the construct. Also, a high-level language may be able to recognize opportunities for special-case optimizations and apply them more uniformly.

### 3. Avalon Language Features

Avalon is a superset of C++. The main features Avalon adds to C++ are transactions and atomic objects, as summarized above, and *servers*.

#### 3.1. Transactions

Transactions are sequences of operations on atomic objects. Transactions are characterized by three properties: *failure atomicity*, *permanence*, and *serializability*. Failure atomicity is the "all-or-nothing" property described above; if a transaction fails to complete (aborts), none of its partial effects become visible. Permanence means that when a transaction has committed, subsequent failures cannot erase its effects. *Serializability* means that the effect of running multiple transactions concurrently must be the same as if they had been run in some serial order. This makes it easier to reason, both formally and informally, about transactions, since each can be thought of as executing without interference. These properties also make it easier to recover a consistent state for a system after a crash: the recovered state should be the same as if all committed transactions executed in their serialization order. We summarize these properties by saying that transactions are *atomic*.

Avalon provides constructs for delimiting transactions (the *start* statement), for starting concurrent transactions (*costart*), and aborting transactions (*undo*).

#### 3.2. Atomic Objects

The atomicity properties of transactions are ensured in Avalon by the implementations of the atomic objects they manipulate. These objects must ensure that enough state is saved to recover from transaction aborts; that committed modifications are written to stable storage to ensure permanence; and that the transactions that execute operations on an object do so in a way that guarantees the serializability of those transactions.

Avalon provides a number of features that make it easy for implementors of classes of atomic objects to meet these requirements. Most of these features are available through inheritance from classes in the Avalon class hierarchy. This hierarchy consists of three classes:

- **RESILIENT** provides **PIN** and **UNPIN** operations that derived classes can use to guarantee failure atomicity. Roughly speaking, these record snapshots of the object before and after modifications. Because these must be called in pairs, Avalon also provides a **pinning block** that brackets its body with a **PIN/UNPIN** pair. The Avalon run-time system guarantees that after a failure, subsequent transactions will observe resilient objects in the state they were in when last **UNPIN**'ed by a transaction that committed.
- **ATOMIC** is a subclass of **RESILIENT**. In addition to **PIN** and **UNPIN**, it provides (protected) operations that derived classes can use to do short-term synchronization of processes concurrently executing operations

at the same object. ATOMIC objects can also provide user-written operations that are invoked by the run-time system when transactions that modified the object commit or abort. This facility can be very useful in the implementation of atomic objects that allow a high degree of transaction-level concurrency.

- DYNAMIC is a subclass of ATOMIC that provides the traditional method of ensuring serializability: strict two-phase read/write locking [4].

### 3.3. Servers

Transactions are ephemeral; they execute for a relatively short time and are finished. Atomic objects, however, may have long lives. Therefore, they are encapsulated in processes called servers. A server is textually similar to a class. The private members of a server (may) include atomic objects. If a server crashes, it is automatically restarted, and its atomic members are recovered in the state seen by the last completed transaction. The public members of a server are operations that can be invoked from other processes by remote procedure calls (RPC's). The server process consists of a loop that listens for an RPC, and forks a lightweight process to handle each request that arrives. Thus, concurrency is allowed within servers. This level of concurrency could be effectively utilized by a shared-memory multiprocessor.

### 3.4. For Further Details...

This section provided only a brief outline of the Avalon language. For more details, interested readers may refer to [5] or [3].

## 4. Avalon as an Exercise in Extending C++

In this section, we will discuss some of the engineering decisions that went into (and are still going into) our implementation of Avalon. We hope that our experience will be of interest to others considering extensions to C++ and other languages.

### 4.1. Levels of Engineering Complexity

When modifying a language, one should obviously attempt to make changes in as simple and modular a way as possible. In our case, our modifications came in three forms, of successively greater complexity:

- *Working within the language.* One of the touted advantages of object-oriented languages and C++ in particular is that classes and operator overloading are supposed to provide the ability to create "virtual languages" customized to particular applications. To a large extent, we found this claim to be well-founded, and used this ability in the class hierarchy described above.
- *Run-time support.* Many of the features of Avalon are provided by a run-time system. Camelot

comprises most of the Avalon run-time system; it provides operating system-level support for transaction management, stable-storage logging, crash recovery, and the like. Avalon-specific elements of the run-time system include a package for dynamically allocating recoverable storage, a server for recording and comparing transaction identifiers for transaction-level synchronization, and a server for registering types and instances of other servers.

- *Language processor.* Avalon is a superset of C++, with a number of new syntactic (and semantic) features. We had to somehow take care of compiling these new constructs into appropriate code. We chose to do this using a preprocessor that translates Avalon into C++.

Implementing a processor for a new language is a complicated undertaking, and we had said previously that we would strive to avoid complication whenever possible. Therefore, we will explain the considerations that went into this decision.

One alternate approach is to implement the new features of Avalon as a macro package. We saw three drawbacks with this approach:

- Using a macro preprocessor would restrict us somewhat in the syntax we could use. While we do not think that syntax is overwhelmingly important, we do believe that a consistent syntax enhances the understandability of a language. Using a separate preprocessor allowed us to choose whatever syntax we wished.
- Macro processors are notorious for producing unintelligible error messages, or worse, for allowing incorrect invocations to pass through to become run-time bugs. The semantics of the new features we provide are complicated enough that many checks must be done to ensure that they are being used correctly. This type of error checking increases the usability of the system by increasing the user's confidence. We also felt that it was important that, as in the C++ preprocessor, all compile-time errors should be detected in the first pass, and later passes (C++ and C, in our case) should function only as code generators. Without this quick feedback, the time required to rid a program of compile-time errors might, to some users, outweigh the advantages of compile-time type-checking.
- Finally, the implementations of many of the features we provide must use information (such as type information) that is available only during the compilation process. It would have been impossible to implement these features using macros.

Given that we had decided to implement a preprocessor, we had the further choice of implementing one from scratch, or of modifying the existing `cfront`. The first option could be rejected easily: it would have been too much work to implement a new preprocessor for the entire Avalon language. Another choice might have been to implement a new preprocessor that handled only the added features of the language. It turned out that the added features interacted extensively enough with the existing features to make this impossible; also, this would have violated the "quick error turnaround" goal. Thus, we decided to work from the existing `cfront`.

For the most part, this decision has been a good one. Working from a stable base, we have been able to concentrate on problems in our extensions, rather than problems in processing C++. However, we have found `cfront` somewhat difficult to work with because it uses a table-driven parser (`yacc`), rather than a recursive descent parser. On a number of occasions we have found it difficult to debug problems with our grammar from the diagnostics provided by `yacc`. With a recursive descent parser, one can at least locate the problem with a debugger. Our other reason for preferring a recursive descent parser is more subtle. In the Avalon preprocessor, we take the parse tree generated by our modified `cfront` and modify it to implement our extensions. These modifications often require the insertion of fairly large amounts of new code. In some cases, we can insert this new code as text, and allow the next pass to compile it, but in others, later code must refer to the generated code for type-checking purposes. For these, we must splice the parse tree for the new code into the existing parse tree. Presently, we must create these parse trees "by hand," by invoking the constructors of the various parse tree node types. It would be much easier if we could construct the string form of the expression or statement we wished to insert, parse that "in place," and insert the resulting subtree into the parse tree. With the current parser, the only syntactic constructs that can be parsed are top-level declarations, so we can not usually use it for this purpose. With a recursive descent parser (or perhaps a differently structured table-driven parser) there could be different routines for parsing declarations, statements, expressions, etc., that we could invoke as needed.

## 5. Our Experience in Using C++

In this section we discuss our experience in using C++ on a moderately large program that uses a large base of existing C code. We have had a number of problems that we thought it would be interesting to share with our readers.

### 5.1. System-wide C++ Compatibility

Avalon programs make extensive use of previously written C code, specifically, the Mach operating system and the Camelot transaction-processing system. We needed to cause the include files that declare the routines we use from these systems to do so in a manner compatible with the C++ argument declaration syntax. We could have done this by making private copies of these files, modifying them, and attempting to track any changes (as is the strategy in the C++ versions of the Unix system include files sent out in the C++ distribution.) This strategy can obviously lead to problems when changes are not accurately and promptly tracked.

The obvious solution to this problem is to maintain one set of include files compatible with both C++ and (non-ANSI standard) C function declaration syntax. Technically, this is simple -- we use a macro developed by Mike Jones at CMU:

```

/* c_args.h:
 * A standard convention for declaring C and C++ functions in header files.
 * Author: Mike Jones, CMU.
 */
#if c_plusplus
#define C_ARGS(arglist) arglist
#else c_plusplus
#define C_ARGS(arglist) ()
#endif c_plusplus

```

After inclusion of this definition, a function `foo` that takes an integer argument `k` may be declared as

```
void foo C_ARGS((int k));
```

Ensuring that these conventions are followed is critical, but difficult, since C++ declarations in C include files are not checked during C compilations. Therefore, inconsistencies are often not noticed until we detect them in our code. We hope to solve part of this problem soon on CMU machines within our department by modifying the standard system include files for C++ compatibility.

## 5.2. Debugging

The Avalon preprocessor is a simple single-thread-of-control program that does no complicated I/O. We are currently debugging the program using `gdb`, the Gnu debugger. For many reasons, this is unsatisfactory. `Gdb`, `dbx`, and most other Unix source-level debuggers, are oriented toward C. Thus, when one runs your program under the debugger, one actually debugs the C code generated by `cfront`. Though this is not as severe a shift as in a debugger such as `adb`, where one debugs the machine code generated by the compiler, it is similar. Several C++ constructs expand into many lines of generated code. Names of variables and structures change drastically, and in some cases in ways that cannot be predicted by looking at the original source (unions are a special problem in this regard.) Debugging C++ code would be greatly aided by C++-oriented debugger.

## 6. Future Directions for Avalon

### 6.1. Writing Test Applications

Once we think we have a working system, our first order of business will be to write some applications to test it. Our first efforts will probably involve the reimplementing of applications being constructed (in C) to test Camelot. These include a conference room reservation system intended to be used by our department, and an ordering system for a department cheese purchasing cooperative. Building these kinds of applications will allow us to gain experience in using our new language constructs, to test and debug the Avalon preprocessor and runtime system, and to explore more generally the topics described below.

## 6.2. Testing Distributed Systems

Testing and debugging a reliable distributed system is a difficult process. In a reliable distributed system, one not only has to worry about the sequential correctness of one's program, but also its interactions with concurrently executing programs, and the possibility of system failure at any time. This makes testing correspondingly harder. Our attitude is that it is impossible to test a complicated distributed system exhaustively enough to give any real confidence in its correctness, especially in application domains with extreme reliability requirements. We must instead rely on methodologies that limit the categories of bugs we can encounter. For instance, it should be possible to test individual Avalon atomic classes in isolation, and have confidence that they will work correctly when combined in a server. That is, once one has satisfied oneself that a class enforces atomicity, one can write a server operation that use instances of that class as if the operation were part of a normal sequential program. Thus, we have at least made testing more tractable. To aid testing at the level of atomic classes, we might investigate ways of automatically generating concurrent tests from a specification of sequential tests of the type [6].

## 6.3. Debugging Distributed Systems

Despite the efforts described above, it seems inevitable that bugs will be found when a distributed system is put together. There will be a need for methods and tools for debugging these systems. These go beyond the scope of conventional debugging aids because they extend across machines, and may involve many processes on each machine. For instance, we might want the ability to single step a thread of control as it migrates from one process to another via an RPC. We might also want the ability to query the system about what threads of control are waiting to perform an operation on an atomic object. We need to develop tools with these and other capabilities for monitoring and debugging these large distributed systems.

## References

- [1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young.  
Mach: A New Kernel Foundation for UNIX Development.  
*In Proceedings of Summer Usenix.* July, 1986.
- [2] Agha, Gul and Hewitt, Carl.  
Concurrent Programming Using Actors.  
*Object-Oriented Concurrent Programming.*  
The MIT Press, Cambridge, MA, 1987.
- [3] D. Detlefs, M. P. Herlihy, and J. M. Wing.  
Inheritance of Synchronization and Recovery Properties in Avalon/C++.  
*In The Proceedings of the 21<sup>st</sup> Hawaii International Conference on System Sciences.* Kailua-Kona, Hawaii, Jan, 1988.
- [4] K. P. Eswaran, James N. Gray, Raymond A. Lorie, and Irving L. Traiger.  
The Notions of Consistency and Predicate Locks in a Database System.  
*Communications of the ACM* 19(11):624-633, November, 1976.
- [5] M. P. Herlihy and J. M. Wing.  
Avalon: Language Support for Reliable Distributed Systems.  
*In The Proceedings of the 17<sup>th</sup> International Symposium on Fault-Tolerant Computing.* Pittsburgh, PA, July, 1987.
- [6] Herlihy, Maurice and Wing, Jeannette.  
*Reasoning About Atomic Objects.*  
Technical Report CMU-CS-87-176, Carnegie-Mellon University, November, 1987.
- [7] C.A.R. Hoare.  
Communicating Sequential Processes.  
*Communications of the ACM* 21(8):666-677, August, 1978.
- [8] Ishikawa, Yutaka and Tokoro, Mario.  
Orient84/K: An Object-Oriented Concurrent Programming Language for Knowledge Representation.  
*Object-Oriented Concurrent Programming.*  
The MIT Press, Cambridge, MA, 1987.
- [9] A. Z. Spector, J. J. Bloch, D. S. Daniels, R. P. Draves, D. Duchamp, J. L. Eppinger, S. G. Menees, D. S. Thompson.  
The Camelot Project.  
*Database Engineering* 9(4), December, 1986.  
Also available as Technical Report CMU-CS-86-166, Carnegie-Mellon University, November 1986.
- [10] Yokote, Yasuhico and Tokoro, Mario.  
Concurrent Programming in ConcurrentSmalltalk.  
*Object-Oriented Concurrent Programming.*  
The MIT Press, Cambridge, MA, 1987.



## 1987 C++ WORKSHOP

### Attendees -- By Name Santa Fe, New Mexico

November 9-10, 1987

Donald Acton  
Univ. of British Columbia  
6356 Agricultural Road  
Computer Science Dept.  
Vancouver, B.C., Canada V6T1W5  
acton@gradf.cf.ubc  
604-228-4327

Jim Adcock  
Hewlett-Packard Company  
Lake Stevens Inst. Div.  
8600 Soper Hill Road - MS E4  
Everett, WA 98205  
hplabs!hp-pcd!hplsla!jlma  
206-335-2138

Kurt Allen  
3M  
3M Center  
207-1W-20, Room 1F-05  
St. Paul, MN 55144  
lhnp4!mmm!allen  
612-736-2366

T. Anthony Allen  
World Bank  
701 18th Street N.W.  
Room J-3-253  
Washington, D.C. 20433  
202-473-8433

Dave Atkins  
AT&T Bell Labs  
19 North Webster St.  
Naperville, IL 60540  
lhpls!dia  
312-355-8738

William Bain, Jr.  
Intel Scientific Computers  
15201 NW Greenbrier Dr.  
Beaverton, OR 97006  
allegro!ogcvax!intelisc!wib  
503-629-7649

Mike Ball  
TauMetric Corp.  
1094 Cudahy Place  
Suite 302  
San Diego, CA 92110  
cod.nosc.mil  
619-275-6381

Tsvi Bar-David  
AT&T Bell Labs  
200 Laurel Ave.  
MT 3F-141  
Middletown, NJ 07748  
lhnp4!mtfml!tsvl  
201-957-6809

Scott Barrett  
PAR Gov. Systems Corp.  
220 Seneca Turnpike  
New Hartford, NY 13413  
uunet!wucs1!wucs2!adiron!scott  
315-738-0600

Penny Bauersfeld  
Xerox  
800 Phillips Road  
MS 139-25D  
Webster, NY 14580  
bauersfeld.wbst@xerox.com  
716-422-9168

Robert Berger  
Datacube, Inc.  
4 Dearborn Road  
Peabody, MA 01960  
berger@datacube.com  
617-535-6644

David Bern  
Institute for Zero Defect Software  
POB 4186  
Warren, NJ 07060  
lzdsw!bern  
201-668-1593

Wayne Bower  
AT&T Bell Labs  
1247 South Cedar Crest Blvd.  
Allentown, PA 18103  
215-770-2629

Lawrence Bowman  
Brigham Young Univ.  
2030 North 500 East  
Provo, UT 84604  
801-378-6581

Todd Bridges  
Sandia National Labs  
Division 2116  
POB 5800  
Albuquerque, NM 87185  
505-844-1858

Denis Brockus  
Emulex Corp.  
3545 Harbor Blvd.  
Costa Mesa, CA 92626  
714-662-5600

Mark Bronson  
Lawrence Berkeley Laboratory  
1 Cyclotron Blvd.  
B 50-245  
Berkeley, CA 94720  
mdbronason@lbl.gov  
415-486-6965

Michael Brown  
SRI International  
333 Ravenswood Avenue  
Menlo Park, CA 94025  
michael@spam.istc.sri.com  
415-859-5104

Daniel Brunier-Coullen  
Matra-Datavision  
Rue De La Perre Du Seu  
Zac De Courtaboeuf  
Les Ulis, France 91940  
69283481

Joe Buhler  
Reed College  
3203 S.E. Woodstock  
Portland, OR 97202  
tektronix!reed!jpb  
503-771-1112

William Bulley  
Applicon/Schlumberger  
4251 Plymouth Road  
Ann Arbor, MI 48106  
313-995-6211

Chris Burghart  
NCAR  
POB 3000  
Boulder, CO 80307  
burghart@rdss.ucar.edu  
303-497-8836

Walter Burkhard  
Univ. of CA - San Diego  
Computer Science and Eng.  
C-014  
La Jolla, CA 92093  
burkhard@ucsd.edu  
619-534-2722

Paul Calder  
Univ. of Stanton  
Center for Integrated Systems  
Campus Dr. Via Palou Rm. 19  
Palo Alto, CA  
calder@uluru  
415-725-3648

Fred Calm  
Schlumberger/Applicon  
829 Middlesex Turnpike  
Billerica, MA 08121  
applicon!calm  
617-671-9975

Richard Campbell  
Univ. of Michigan  
1075 Beal Avenue  
Room 142 CCWest  
Ann Arbor, MI 48109  
ric@rioja.cc.umich.edu  
313-763-6188

Roy Campbell  
Univ. of Illinois  
Dept. of Computer Science  
1304 West Springfield  
Urbana, IL 61801  
roy.b.cs.uiuc.edu  
217-333-0215

Tom Cargill *cargill*  
AT&T Bell Labs  
Murray Hill, NJ 07974  
research!tac  
201-582-7393

John Carolan  
Glockenspid  
19 Belvedere Place  
Dublin 1, Ireland  
mcvax!einode!puschl!john  
011+353+1+364515

Dale Carstensen  
Los Alamos Nat'l Labs  
Group C-3  
MS B265  
Los Alamos, NM 87545  
dlc@lanl.gov lanl!dlc

Norman Cheung  
Fujitsu America  
3055 Orchard Drive  
Building 2  
San Jose, CA 95134-2017  
408-432-1300

Norman Chibuk  
J.S. Norad Inc.  
316 Erskine Road  
Stanford, CT 06903  
203-322-0320

Yun-Sook Chol  
Hewlett-Packard Company  
5301 Stevens Creek Blvd.  
Santa Clara, CA 95052  
hplabs!hpsoda!hpsc!sookie  
408-553-3304

Chuck Clanton  
Aratar  
220 Downey Street  
San Francisco, CA 94117  
sun!plato!chac  
415-681-4140

Laurence Clark  
AT&T  
229 W. 7th Street  
7th Floor  
Cincinnati, OH 45202  
513-784-3156

Susan Coatney  
Univ. of Southern CA  
Information Sciences Institute  
4676 Admiralty Way  
Marina Del Rey, CA 90292-6695  
coatney@venera.isi.edu  
213-822-1511

James Coggins  
Univ. of North Carolina  
New West Hall 035A  
Computer Science Dept.  
Chapel Hill, NC 27514  
919-962-1738

Mike Coleman  
Tektronix, Inc.  
POB 500  
Delivery Station 58-639  
Beaverton, OR 97077  
mc@videovax.pv.tek.com  
503-627-2358

Patrick Conley  
Abraxaf Software Inc.  
7033 S.W. Macadam Ave.  
Portland, OR 97219  
503-244-5253

Al Conrad  
Univ. of CA - Santa Cruz  
CIS - Board  
Santa Cruz, CA 95060  
conrad@saturn.ucsc.edu  
408-429-2370

Donald Courtney  
Digital Equipment Corp.  
110 Splitbrook Road  
Nashua, NH 03062  
603-881-2108

Fred Cox  
Athena Systems, Inc.  
4 Main Street  
G-1  
Los Altos, CA 94022  
sun!imagen!atari!daisy!athena  
415-941-4818

Gary Craig  
US West  
Advanced Technology Division  
6200 South Quebec Suite 301  
Englewood, CO 80111  
303-930-5340

Edward Currie  
Lifeboat Associates  
55 South Broadway  
Tarrytown, NY 10591  
914-332-1875

Robin Das  
Lotus Development Corp.  
55 Cambridge Parkway  
Room 9044  
Cambridge, MA 02142  
617-577-8500

Judy DesHarnais  
USENIX Association  
POB 385  
16951 Pacific Coast Hwy  
Sunset Beach, CA 90742  
usenix!judy  
213-592-1381

David Dettles  
Carnegie Mellon Univ.  
Computer Science Dept.  
Schenley Park  
Pittsburgh, PA 15217  
dld@g.cs.cmu.edu  
412-268-3778

Stephen Dewhurst  
AT&T  
190 River Road  
E-324  
Summit, NJ 07901  
attunix!scd  
201-522-6083

Mike Dieter  
Apollo Computer, Inc.  
330 Billerica Road  
Chelmsford, MA 01824  
617-256-6600

Jim Dimino  
US West Advanced Technologies  
6200 S. Quebec  
Suite 260  
Englewood, CO 80111  
303-930-2507

Thomas Doeppner  
Brown Univ.  
Dept. of Computer Science  
Providence, RI 02912  
(lhnpp4,decvax)!brunix!twd  
401-863-1834

John Donnelly  
USENIX Association  
Exhibit Office  
5398 Manhattan Circle  
Boulder, CO 80303  
usenix!john  
303-499-2600

Paul Dubois  
Lawrence Livermore Nat'l Lab  
POB 5508  
MS L-471  
Livermore, CA 94550  
415-422-4237

Gordon Durand  
Cadnetix Corp.  
5757 Central Ave.  
Boulder, CO 80301  
cadnetix!gad  
303-444-8075

Bruce Eckel  
Univ. of Washington  
School of Oceanography, WB-10  
Physical Oceanography Dept.  
Seattle, WA 98195  
206-543-6455

Carl Ellis  
Oregon Software, Inc.  
6915 SW Macadam Ave.  
Portland, OR 97219  
...!tektronix!oresoft!carl  
503-245-2202

Michelle Emanuel  
AT&T Bell Labs  
Crawfords Corner Road  
Room 2B 502  
Holmdel, NJ 07733  
lhnpp4!ark1!1mte  
201-949-1292

Glenn Engel  
Hewlett-Packard Company  
Lake Stevens Inst. Div.  
8600 Soper Hill Road - MS E4  
Everett, WA 98205  
hplabs!hp-pcd!hplsia!glenn  
206-335-2066

Eric Feigenson  
Persoft, Inc.  
465 Science Drive  
Madison, WI 53711  
608-273-6000

Paul Fillinich  
AT&T Bell Labs  
190 River Road  
Room 3-206  
Summit, NJ 07901  
attunix!paulf  
201-522-6664

David Floodpage  
Lotus Development Corp.  
55 Cambridge Parkway  
Cambridge, MA 02142  
617-577-8500

Peter Ford  
Los Alamos Nat'l Labs  
Center for Nonlinear Studies  
MS B258  
Los Alamos, NM 87545  
peter@cardinal.lanl.gov  
505-665-0058

David Forslund  
Los Alamos Nat'l Labs  
P.O. Box 1663  
MS E531  
Los Alamos, NM 87545  
505-665-1907

Bruce Frederiksen  
NCR Corp.  
SE-Retail, SER-r 22  
1700 So. Patterson Blvd.  
Dayton, OH 45479  
b.frederiksen@dayton.ncr.com  
513-865-8035

Ken Friedenbach  
Apple Computer, Inc.  
10500 North DeAnza Blvd.  
MS 27-E  
Cupertino, CA 95014  
408-973-2539

Ken Fuhrman  
Ampex Corp.  
10604 W. 48th Ave.  
Wheat Ridge, CO 80228  
(hao!udenva!avsd)!ascvax!ken  
303-423-1300

Kent Fuka  
Convex Computer Corp.  
POB 833851  
Richardson, TX 75083  
lhnpp4!convex!fuka  
214-952-0200

Phillippe Gautron  
INRIA  
B.P. 10505  
78153 Le Chesnay  
C'edex, France  
mcvax!lnria!corto!gautron  
+33 (1)39-63-53-25

Bill Gerger  
AT&T  
11900 N. Pecos  
Denver, CO 80234  
druhl!gerg  
303-538-4281

Albert Gettler  
US West  
Advanced Technology Div.  
6200 South Quebec, Suite 301  
Englewood, CO 80111  
infoswix!al  
303-930-2507

Jeff Glover  
Tektronix, Inc.  
POB 1000  
D/S 61-040  
Wilsonville, OR 97070  
jeffg@tekecs.tek.com  
503-685-2207

Keith Gorlen  
National Institutes of Health  
Bg. 12A, Room 2017  
Bethesda, MD 20892  
usenix!nih-csl!keith  
301-496-5363

Bruce Graham  
Oregon Software, Inc.  
6915 S.W. Macadam  
Portland, OR 97219  
tektronix!reed!oresoft!bruce  
503-245-2202

Michaela Guiney  
Hewlett-Packard Company  
5301 Stevens Creek Blvd.  
Santa Clara, CA 95052  
hplabs!hpdte!mg  
408-553-2985

Roy Hall  
Wavefront Technologies, Inc.  
530 E. Montecito Street  
R and D  
Santa Barbara, CA 93103  
ucbvax!ucsbcs!wavefro!roy  
805-962-8117

Michael Hannah  
Sandia National Labs  
Division 2614  
POB 5800  
Albuquerque, NM 87185  
505-846-3459

Gilbert Hansen  
Convex Computer Corp.  
701 Plano Road  
Richardson, TX 75081  
hansen@convex  
214-952-0348

Kathleen Harris  
Hewlett-Packard Company  
3404 E. Harmony Road  
Fort Collins, CO 80526  
(ihnp4,hplabs)!hpfcla!kah  
303-229-3167

Ken Harris  
Unico, Inc.  
3725 Nicholson Road  
Franksville, WI 53126  
414-886-5678

Mitch Harter  
Microsoft Corp.  
16011 NE 36th Way  
POB 97017  
Redmond, WA 98073-9717  
206-882-8080

Sassan Hazeghi  
Peritus International, Inc.  
POB 4526  
Stanford, CA 94305  
decwrl!oliveb!perilus!sasson  
408-725-0882

Paul Hegarty  
Stanford Univ.  
Sweet Hall, 3rd Floor  
Stanford, CA 94305-3091  
hegarty@score.stanford.edu  
415-723-5264

Dave Hinman  
MASSCOMP  
1 Technology Park  
Westford, MA 01886  
decvax,ihnp4!masscomp!hinman  
617-692-6200

Leland Hoover  
Anasazi, Inc.  
7500 North Dreamy Draw Drive  
Suite 120  
Phoenix, AZ 85020  
602-870-3330

Bill Hopkins  
AT&T  
11900 N. Pecos Street  
Denver, CO 80234  
ihnp4!allegra!druhi!weh  
303-538-4944

Tai-Yuan Hou  
Siemens  
105 College Road East  
Princeton, NJ 08540  
609-734-6563

Donna Ho  
Hewlett-Packard Company  
3500 Deer Creek Road  
Bldg. 26U  
Palo Alto, CA 94304  
!hplabs!hpcea!hpcea!hpceea!ho  
415-857-3581

Andrew Hume  
AT&T Bell Labs  
600 Mountain Ave.  
Room 2C-471  
Murray Hill, NJ 07974  
research!andrew  
201-582-6262

Michael Hunter  
IBM Corp.  
11400 Burnet Road  
Dept. D98 Bldg. 803  
Austin, TX 78758  
512-823-4058

Reggie Hutcherson  
SRI International  
333 Ravenswood Avenue  
Menlo Park, CA 94025  
hutch@spam.iste.sri.com  
415-859-2077

Cheryl Hyslop  
GeoVision Corp.  
1600 Carling Ave.  
Suite 350  
Ottawa, Ont., Canada K1Z8R7  
613-722-9518

John Isner  
AT&T Bell Labs  
184 Liberty Corner Road  
Room 4NE05  
Warren, NJ 07060  
attunix!isner  
201-580-8074

Fumiko Ito  
Fujitsu Limited  
2-1 Software Technology  
1015 Kamiotanaka  
Nakahara-ku, Kawasaki, Japan 211  
011-81-044-777-1111

Curtis Johnson  
NCR Corp.  
Systems Engineering - San Diego  
9900 Old Grove Road  
San Diego, CA 92131-1685  
!ncr-sd!se-sd!curtis  
619-693-5752

Dave Jones  
Megatest Corp.  
880 Fox Lane  
San Jose, CA 95131  
megatest!djones@glacier.arpa  
408-437-9700

David Jones  
Microsoft Corp.  
16011 NE 36th Way  
POB 97017  
Redmond, WA 98073-9717  
206-882-8080

George Kakatsakis  
SAE, Inc.  
2120 Miller Drive  
Longmont, CO 80501  
boulder!stan!georgek  
303-772-3400

Deepak Kanwar  
AT&T Bell Labs  
6200 E. Broad Street  
Room 2B-239A  
Columbus, OH 43213  
cb!cbsck!kan  
614-860-5891

Stacey Keenan  
AT&T Bell Labs  
190 River Road  
SF F315  
Summit, NJ 07901  
attunix!stacey  
201-522-6149

Denis Kertz  
AT&T Bell Labs  
200 Park Plaza  
Room 2F-527  
Naperville, IL 60566  
ihnp4!ihlpg!drk  
312-416-7073

Anton Kirchner  
AT&T  
8200 E. Maplewood Ave.  
Room 110  
Englewood, CO 80111  
!hnp4!ncsc5!ahk  
303-850-8719

Peter Kirsils  
AT&T  
11900 N. Pecos Street  
Denver, CO 80234  
lhnp4!druhl!kirsils  
303-538-4061

Bob Klein  
Solutions Are Everything Inc.  
2120 Miller Drive  
Longmont, CO 80501  
hao!boulder!stan!rmk  
303-772-3400

Gary Klimowicz  
Sequent Computer Systems  
15450 SW Koll Parkway  
Beaverton, OR 97006  
sequent!gak  
503-626-5700

Reed Koch  
Microsoft Corp.  
16011 NE 36th Way  
POB 97017  
Redmond, WA 98073-9717  
206-882-8080

Andrew Koenig  
AT&T Bell Labs  
184 Liberty Corner Road  
Room 4N-R12  
Warren, NJ 07060  
attunix!ark  
201-580-4883

Anastasios Kontogloraos  
AT&T Bell Labs  
Crawfords Corner Road  
3M 324  
Holmdel, NJ 07733  
201-949-7184

James Kuehn  
Supercomputing Res. Ctr  
4380 Forbes Boulevard  
Lanham, MD 20706  
kuehn@super.org (csnet)  
301-731-3746

Douglas Landauer  
Sun Microsystems, Inc.  
2550 Garcia Ave.  
Mountain Ave., CA 94043  
landaver@sun.com  
415-960-1300

Ronald Lee  
SRI International  
333 Ravenswood Avenue  
Menlo Park, CA 94025  
rlee@spam.istc.sri.com  
415-859-4630

Ernest Leggett  
Teknekron Infoswitch Corp.  
1784 Firman Drive  
Richardson, TX 75081  
!hnp4!infoswx!ewl  
214-644-0570

Ron Levine  
Dorian Research, Inc.  
808 Burlway Road  
No. 207  
Burlingame, CA 94020  
well!levine  
415-342-2229

Serge Limondin  
Automatix, Inc.  
1000 Technology Park Drive  
BillERICA, MA 01821  
617-667-7900

Moses Ling  
AT&T Bell Labs  
23-02 Fontana Heights  
39 Mount Sinal Rise  
Singapore, China 1027  
lhnp4!zplnt!mml  
011-654673270

Mark Linton  
Univ. of Stanford  
Center for Integrated Systems  
Campus Drive Via Palou Room 213  
Palo Alto, CA  
415-725-3717

Stanley Lippman  
AT&T  
Room E-330  
190 River Road  
Summit, NJ 07901  
lhnp4!attunix!stan  
201-522-6252

Steven Loving  
Stanford Univ.  
Sweet Hall, 3rd Floor  
Stanford, CA 94305-3091  
loving@jessica.stanford.edu  
415-723-9214

Joe MacDougald  
Apple Computer, Inc.  
10500 N. De Anza Blvd.  
MS 27 AQ  
Cupertino, CA 95014  
joemac@apple.com  
408-973-6426

Glenn Machin  
Sandia National Labs  
Division 2113  
POB 5800  
Albuquerque, NM 87185  
505-846-3459

Frank Maestas  
Los Alamos Nat'l Labs  
POB 1663  
C-8/MS B294  
Los Alamos, NM 87545  
fam@lanl.gov  
505-667-1013

Suzanne Maimendler  
AT&T Bell Labs  
Crawfords Corner Road  
HO-3K-338  
Holmdel, NJ 07733  
201-949-0665

Michael Marks  
Lawrence Livermore Nat'l Lab  
POB 808  
L-60  
Livermore, CA 94550  
415-423-6554

Dean Mason  
SRI International  
333 Ravenswood Avenue  
Menlo Park, CA 94025  
mason@spam.istc.sri.com  
415-859-5873

Mark Mathieu  
Hewlett-Packard Company  
POB 301  
Loveland, CO 80539  
hplabs!hpfcla!hplvla!mathieu  
303-667-2804

M. Haytham Matthews  
IRMAI  
1119 Boynton Ave.  
Bronx, NY 10472  
212-893-3048

Patricia Max  
Los Alamos Nat'l Labs  
POB 1663  
Los Alamos, NM 87544  
pam@sonny.lanl.gov  
505-667-7280

Oliver McBryan  
Univ. of Colorado  
Dept. of Computer Science  
Campus Box 430  
Boulder, CO 80309  
mcbryan@boulder.colorado.edu  
303-492-3898

William McIntosh  
Wilson Learning Corp.  
2009 Pacheco Street  
Santa Fe, NM 87505  
505-471-6500

Carol Meier  
4222 Corriente Pl.  
Boulder, CO 80301  
303-938-8031

Karen Meyer-Arendt  
Solutions Are Everything, Inc.  
2120 Miller Drive  
Longmont, CO 80501  
hao!boulder!stan!arendt  
303-772-3400

Richard Meyers  
Apple Computer, Inc.  
20525 Mariani Ave.  
MS 27-E  
Cupertino, CA 95014  
408-973-3285

Phillip Miller  
Century Computing, Inc.  
1100 West Street  
Laurel, MD 20707  
301-953-3.30

William Miller  
Software Dev. Tech., Inc.  
375 Dutton Road  
Sudbury, MA 01776-2509  
...!genrad!mrst!sdtt!wmn  
617-443-5779

Walt Moleski  
NASA/GSFC  
Greenbelt Road  
ATTN: Code 522.1  
Greenbelt, MD 20771  
301-286-7633

Barbara Moo  
AT&T Bell Labs  
184 Liberty Corner Road  
POB 4908  
Warren, NJ 07060  
attunix!bem  
201-580-4056

Joseph Moran  
Sun Microsystems, Inc.  
2550 Garcia Ave.  
Mountain View, CA 94041  
mojo@sun.com  
415-691-7292

David Morein  
proCASE  
3130 De La Cruz Blvd.  
Suite 100  
Santa Clara, CA 95054  
ucbvax!tolerant!procas!dsm  
408-727-0714

Robert Murray  
AT&T Bell Labs  
Room 4N-W09,  
184 Liberty Corner Road  
Warren, NJ 07060  
ihnp4!kaiser!rbm  
201-580-5742

Mark Nettleingham  
AT&T  
4200 Commerce  
Suite 105  
Lisle, IL 60532  
ihnp4!lll!markn  
312-983-4475

Peter Nicklin  
Hewlett-Packard Company  
5301 Stevens Creek Blvd.  
MS 53U/88  
Santa Clara, CA 95052  
nicklin%hpdtc@hplabs.hp.com  
408-553-2982

Ramakrishnan Niranjan  
Mentor Graphics Corp.  
8500, SW Creekside Place  
Beaverton, OR 97005  
503-626-7000

Kevin Nolan  
QTC  
8700 S.W. Creekside Place  
Suite D  
Beaverton, OR 97005  
tektronix!sequent!qtc!kevin  
503-626-3081

Tim O'Konski  
Hewlett-Packard Company  
3500 Deer Creek Road  
Bldg. 26U  
Palo Alto, CA 94304  
hplabs!hpcea!hpceasea!tim  
415-857-3999

Neil Ostrove  
AT&T Bell Labs  
6200 E. Broad Street  
Columbus, OH 43213  
cdbkcl!ost  
614-860-4500

Wallace Owen  
Unisys Corp.  
4674 Del mMonte  
San Diego, CA 92107  
cod.nosc.mil!owen  
619-224-9105

Sam Palmer  
Memory Data Software, Inc.  
3932 Cerritos Ave.  
Long Beach, CA 90807  
213-424-3722

Bob Phillips  
Oregon Software, Inc.  
6915 SW Macadam Ave.  
Portland, OR 97219  
...!tektronix!oresoft!bob  
503-245-2202

Michael Pollock  
The World Bank  
701 18th Street N.W.  
Room J-3-253  
Washington, D.C. 20433  
202-473-8433

John Quarterman  
Texas Internet Consulting  
701 Brazos Austin Centre  
Suite 500  
Austin, TX 78701-3243  
jsq@longway.tlc.com  
512-320-9031

Kectrin Radigan  
AT&T Bell Labs  
6200 E. Broad Street  
Room 2B-239A  
Columbus, OH 43213-1569  
cb!cbseck!klr  
614-860-2014

Mark Rafter  
Warwick Univ.  
Computer Science Dept.  
Coventry, England CV47AL  
uucp...!mcvax!warwick!rafter  
44-203-523364

Ragu Raghavan *Raghavan*  
Mentor Graphics Corp.  
8500 SW Creekside Place  
Beaverton, OR 97005  
mntgfx!pdx.mentor.com!ragur  
503-626-7000

Chltraleka Ramanujan  
Univ. of Michigan  
Computing Center  
1075 Beal Avenue  
Ann Arbor, MI 48105  
ramanujan@um.cc.umich.edu  
313-763-6053

Douglas Rand  
Prime Computer  
500 Old Connecticut Path  
MS 10C-17  
Framingham, MA 01701  
doug@eddie.mit.edu  
617-879-2960

Ted Reed  
Los Alamos Nat'l Labs  
C-6  
MS B-272  
Los Alamos, NM 87545  
tnr@lanl.gov  
505-667-0935

Jane Resen  
West. Area Power Admin.  
200 4th Street SW  
Huron, SD 57350  
605-886-5793

Michael Rickabaugh  
Digital Equipment Corp.  
77 Reed Road  
MS HL02-2/H13  
Hudson, MA 01749  
617-568-5139

Jean-Paul Rigault  
Applied Mathematics Center  
Ecole Des Mines  
Sophia Antipolis  
Valbonne, France 06560  
lnria.lnria.fr!ceris!jpr  
33-93-95-75-75

Craig Robertson  
Teradyne, Inc.  
30801 Agoura Road  
Agoura Hills, CA 91301  
818-991-2900

Eric Roberts  
DEC - SRC  
130 Lytton Ave.  
Palo Alto, CA 94301  
robert@src.dec.com  
415-853-2111

Stephen Rogers  
Decision Software  
51 Spinelli Place  
Cambridge, MA 02138  
uunet!mrmarx!sdr  
617-576-7800

Frank Rosbach  
Codex Corp.  
20 Cabot Blvd.  
C1-65  
Mansfield, MA 02048  
617-364-2000

John Rose  
Thinking Machines Corp.  
245 First Street  
Cambridge, MA 02142  
lhnp4!think!rose  
617-876-1111

Paul Roush  
Teradyne, Inc.  
30801 Agoura Road  
Agoura Hills, CA 91301  
818-991-2900

Whitney Rusty  
Oregon Software, Inc.  
6915 SW Macadam Ave.  
Portland, OR 97219  
...!tekonix!oresoft!rusty  
503-245-2202

Daniel Sanderson  
Digital Equipment Corp.  
301 Rockrimmon Blvd. So.  
MS CX01-2/N23  
Colorado Springs, CO 80919  
303-548-2072

Sam Sands  
Hewlett-Packard Company  
3404 E. Harmony Road  
MS 7  
Fort Collins, CO 80525  
sam%hpfcpl@hplabs.hp.com  
303-229-2757

Don Sawtelle  
Ampex Corp.  
401 Broadway  
MS 3-21  
Redwood City, CA 94063  
ucbvax!avsd!sawtelle  
415-367-2854

Deborah Scherrer  
mt Xlnu  
2560 Ninth St.  
Suite 312  
Berkeley, CA 94710  
uunet!mtxlnu!scherrer  
415-644-0146

Edward Schiebel  
AT&T Bell Labs  
Whippany Road  
Room 4C-352  
Whippany, NJ 07981  
201-386-3416

David Schilder  
West. Area Power Admin.  
200 4th Street SW  
Huron, SD 57350  
605-886-5793

Michele Schirru  
Los Alamos Nat'l Labs  
C-6  
MS - B272  
Los Alamos, NM 87545  
mes@lanl.gov  
505-667-4119

Dan Schuh  
Univ. of Wisconsin  
1210 West Dayton  
Computer Science Dept.  
Madison, WI 53706  
uwxvax!schuh  
608-262-7892

Gerrie Schults  
Hewlett-Packard Company  
3404 East Harmony Road  
MS 7  
Fort Collins, CO 80525  
gerrie%hpfcpl@hplabs.hp.com  
303-229-3709

Jerry Schwarz  
AT&T Bell Labs  
600 Mountain Ave.  
Room 3C 536 B  
Murray Hill, NJ 07974  
ulysses!jss  
201-582-5406

Donn Seeley  
Univ. of Utah  
Computer Science Dept.  
Merrill Engineering Bldg.  
Salt Lake City, UT 84112  
donna@cs.utah.edu  
801-581-5668

Jere Shank  
AT&T Bell Labs  
Whippany Road  
Room 4C340A  
Whippany, NJ 07981  
vilya!jere  
201-386-3438

Marc Shapiro  
INRIA  
B.P. 10505  
78153 Le Chesnay  
C'edex, France  
mcvax!inria!shopiro  
+33 (1)39-63-53-25

Vinnie Shelton  
Decision Software  
51 Spinelli Place  
Cambridge, MA 02138  
uunet!mrmarx!acs  
617-576-7112

Jonathan Shopiro  
AT&T Bell Labs  
600 Mountain Ave.  
Room 2C-576  
Murray Hill, NJ 07974  
research!shopiro  
201-582-4179

Richard Sitze  
TCI Software Research  
1190 B Foster Road  
Las Cruces, NM 88001  
sitze@nmsu  
505-522-4600

Alan Snyder  
Hewlett-Packard Company  
POB 10490  
Palo Alto, CA 94303-0971  
snyder@hplabs.hp.com  
415-857-8764

Kathy Stark  
AT&T Bell Labs  
190 River Road  
Summit, NJ 07901  
...attunix!stark  
201-522-6267

Daniel Stearns  
Cal Poly San Luis Obispo  
Computer Science Dept.  
San Luis Obispo, CA 93407  
805-756-7182

Bjarne Stroustrup  
AT&T Bell Labs  
Murray Hill, NJ 07974  
research!bs  
201-582-7393

John Tarbotton  
J & J Engineering  
1164 Mentone Street  
Grover City, CA 93433  
805-481-6847

Craig Taylor  
Sun Microsystems, Inc.  
2550 Garcia Ave.  
A4-49  
Mountain View, CA 94043  
craigtaylor@sun.com  
415-354-4704

Michael Tebo  
Sandia National Labs  
Division 2113  
POB 5800  
Albuquerque, NM 87185  
505-844-7192

Tim Tessin  
Lawrence Livermore Nat'l Lab  
P.O. Box 808 L-542  
Livermore, CA 94550  
tjt@111-lls.arpa  
415-423-4560

Tom Thomas  
Wilson Learning Corp.  
2009 Pacheco Street  
Santa Fe, NM 87505  
505-471-6500

Michael Tiemann  
MCC  
3500 West Balcones Center Drive  
Austin, TX 78759  
tiemann@mcc.com  
512-338-3761

Yarko Tymciurak  
Univ. of Arizona  
Dept. of Physics Bldg. 81  
Room PAS - 368  
Tucson, AZ 85721  
arizona!yak!yak  
602-621-2884

Jacques VanGorp  
NCAR  
POB 3000  
FOF  
Boulder, CO 80307  
vangorp@rdss.uucp  
303-497-8843

Mike Vermeulen  
Hewlett-Packard Company  
3404 East Harmony Road  
Fort Collins, CO 80525  
hplabs!hpfert!meu  
303-229-4462

James Waldo  
Apollo Computer, Inc.  
330 Billerica Road  
Chelmsford, MA 01824  
decvax!apollo!waldo  
617-256-6600

Stan Webb  
EDS - Research  
2155 Louisiana N.E.  
Suite 9100  
Albuquerque, NM 87110  
505-883-6931

Waldo Wedel  
NBI, Inc.  
POB 9001  
3450 Mitchell Lane  
Boulder, CO 80301  
nbires!wedel  
303-938-2923

Dave Weil  
Microsoft Corp.  
16011 NE 36th Way  
POB 97017  
Redmond, WA 98073-9717  
206-882-8080

Pierre Wellner  
Xerox  
1350 Jefferson Road  
Bldg. 801-31A  
Rochester, NY 14623  
wellner;henr801c;xerox.com  
716-427-1000

Gordon Whitten  
Microsoft Corp.  
16011 NE 36th Way  
POB 97017  
Redmond, WA 98073-9717  
206-882-8080

David Wonnacott  
AT&T  
P.O. Box 1000  
Corporate Education Center  
Hopewell, NJ 08525  
!hnp4!rz3bb!davew  
609-639-4543

John Yakemovic  
Tridom Corp.  
840 Franklin Court  
Suite E  
Marietta, GA 30067  
gatech!dscatl!tridom  
404-426-4261

K. C. Burgess Yakemovic  
NCR Corp.  
Systems Engineering - Retail  
37 Executive Park Drive, N.E.  
Atlanta, GA 30329  
kcby@seradg.Dayton.NCR.com  
404-982-8618

David Yost  
Grand Software, Inc.  
8464 Kirkwood Drive  
Los Angeles, CA 90046  
{uunet,attmail}!grand!day  
213-650-1089

# USENIX Association Services and Benefits

The USENIX Association is a not-for-profit organization of individuals and institutions with an interest in UNIX® and UNIX-like systems and the C programming language. It is dedicated to fostering the development and communication of research and technological information and ideas pertaining to UNIX and UNIX-related systems. The Association sponsors workshops and semiannual technical meetings, produces and distributes a bimonthly newsletter called *login:*, publishes a quarterly technical journal called *Computing Systems*, and serves as coordinator of a software exchange via its "Software Distribution Tapes."

The Association was formed in 1975 and incorporated in 1980 to meet the needs of UNIX users and system maintainers who met periodically to discuss problems and exchange ideas concerning UNIX. It is governed by a Board of Directors elected biennially.

There are four classes of membership in the USENIX Association, differentiated primarily by the fees paid and services provided. There is a detailed description of the membership classes on the reverse side.

The USENIX Association offers several services to its members:

- Mailing of the newsletter *login:*;
- Mailing of technical journal *Computing Systems*;
- Offering of various UNIX publications and technical information for purchase;
- Presentation of technical meetings twice a year and single-topic workshops periodically;
- The right to order 4.3BSD UNIX Manuals;
- A discount on the meeting registration fee;
- The right to vote on matters affecting the Association, its bylaws, and in the election of its directors and officers.

Corporate, Educational, and Supporting Members are also offered the right to receive member-contributed software following upon completion of appropriate release forms.

Further information on membership and services and membership applications can be obtained from the Association Office:

USENIX Association  
P.O. Box 2299  
Berkeley, CA 94710  
(415) 528-8649  
{ucbvax,uunet}!usenix!office

# USENIX Association Membership Application

Membership is by Calendar Year

*Please type or print*

☐ New      ☐ Renewal - Member number from mailing label: \_\_\_\_\_

Member Name: \_\_\_\_\_

Address: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

Phone: \_\_\_\_\_

uucp network address: *uunet!* \_\_\_\_\_

Individual, Corporate, and Supporting categories are all open to either institutions or individuals.  
Membership fees are:

☐ \$ 40 Individual

☐ \$ 15 Student (full-time)

*With copy of student I.D. card*

☐ \$ 275 Corporate

☐ \$125 Educational Institution

☐ \$1000 Supporting

☐ Check enclosed: \$ \_\_\_\_\_

*Payments must be in US dollars payable on a US bank*

☐ Purchase order enclosed; invoice required

☐ Check if you do NOT want your name and address made available to other members.

☐ Check if you do NOT want your name and address made available for commercial mailings.

